UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Trabajo de fin de máster

Aceleración de algoritmos CFD en sistemas heterogéneos PC-FPGA con paradigmas de paralelismo.

Autor:
Ismael Gómez García

Director:
Sergio López Buedo

Madrid, 2013

Índice

1. Introducción

1.1 Motivación

En la actualidad, los algoritmos CFD son una herramienta muy utilizada en el mundo de la aeronáutica, ya que permiten abordar una gran variedad de problemas físicos sin solución analítica de una forma computacional, aportando soluciones software que sirven después como guía para posibles diseños experimentales. Hay que tener en cuenta que, en la actualidad, las compañías aeronauticas utilizan principalmente ensayos de tunel de viento para validar sus diseños industriales, procedimiento de un coste muy elevado. Por este motivo, existe un gran interés en el desarrollo de métodos computacionales que permitan reducir el número de ensayos de este tipo, ahorrando así gran parte del coste.
Este tipo de métodos ya existen, pero, debido a la gran complejidad de este tipo de algoritmos, y a la enorme cantidad de datos que manejan, el coste computacional de estas simulaciones es muy grande, motivo por el cual existe un gran interés en el estudio de tecnologías y métodos que permitan efectuar estas simulaciones en menos tiempo.
Paralelamente a esta cuestión, la paralelización por hardware, y en particular el uso de FPGAs para este propósito es una disciplina emergente, que arroja cada vez más resultados que invitan a pensar que este tipo de técnicas, además de sus muchas aplicaciones ya consolidadas, podrían servir para acelerar algoritmos que cumplan unas características determinadas (en particular, que admitan un elevado grado de procesamiento paralelo).
Enlazando con lo anterior, el campo de las FPGAs ofrece una oportunidad de paralelizar los algoritmos CFD, aportando así una solución novedosa que podría traspasar los actuales límites en este tipo de tecnología, dados por las limitaciones de las máquinas que se utilizan a la hora de ejecutar estos algoritmos (a menudo clusterizados).

1.2 Objetivo del trabajo

Mediante este trabajo de fin de máster, se ha pretendido desarrollar una plataforma hardware que ejecute un solver CFD básico previamente adaptado, con el objetivo de lograr acelerarlo, obteniendo así un ejemplo que muestre la validez de la tecnología FPGA a la hora de resolver este tipo de problemas.
En el presente documento se explica el trabajo realizado paso por paso:
En primer lugar (sección 2), un breve análisis del estado del arte de los CFDs, así como del uso de tecnología hardware para la aceleración de este tipo de algoritmos. Este análisis inicial es el que ha permitido conocer el trabajo ya realizado en este área, pudiendo así elegir un enfoque adecuado para el trabajo.
A continuación, en la sección 3, se presenta la metodología de trabajo con Vivado HLS, una herramienta que será fundamental en el desarrollo del proyecto.
En la sección se presenta el diseño conceptual adoptado, explicando las principales dificultades que planteaba el problema, y las soluciones de diseño adoptadas para solventar estas dificultades.
Tras esto, en las secciones 5 y 6, se explica paso a paso el proceso de implementación que se ha llevado a cabo, indicando el punto de partida del trabajo, así como los motivos que han llevado a cada una de las decisiones a nivel de implementación.
Finalmente, en las secciones 7, 8 y 9, se presentan los resultados que se han obtenido, las conclusiones, y el posible futuro de este tipo de aplicaciones.

2. Estado del arte

En esta sección se presenta un breve resumen acerca del estado del arte de las tecnologías sobre las que se ha trabajado en este proyecto. Este análisis es fundamental a la hora de orientar el trabajo, de modo que este resulte original, y permita aportar alguna mejora al realizado con anterioridad en el área sobre la que estamos trabajando.

2.1 Algoritmos CFD

En la actualidad, los algoritmos CFD (Computational Fluid Dynamics) son una herramienta muy utilizada dentro del mundo industrial, en particular en el de la aeronaútica, donde gozan de un amplio campo de aplicación como herramientas de simulación.
Este tipo de métodos tratan problemas físicos de dinámica fluidos mediante una aproximación discreta, lo que permite resolver numéricamente problemas cuya solución analítica es desconocida (como las ecuaciones de Euler o las de Navier-Stokes).
Normalmente, este tipo de problemas están definidos en un dominio finito, cuya forma varía en función del problema. La situación típica que presentan los problemas que hemos utilizado como ejemplo en el presente trabajo es un dominio tridimensional, con el ala de un avión en el centro. En los bordes, se toma una frontera cuya distancia respecto del ala es muy grande, de modo que el borde exterior del dominio no influya en el comportamiento del fluido cerca del ala (que es lo que típicamente nos interesa cuando estudiamos este tipo de problemas).

Como ya se ha mencionado, el tratamiento que se da a este tipo de problemas es aproximado, ya que no existe solución analítica para los mismos. Esto conlleva una dificultad, y es que al discretizar se pierde precisión en las soluciones, por lo que pueden aparecer errores numéricos que provoquen que el método no converja. Para evitar esta dificultad, se toman mallas de muchos puntos (varios millones de ellos), lo que deriva en un segundo problema: que la ejecución de este tipo de métodos se vuelve muy costosa en tiempo de computación. Por este motivo, hay un gran interés en posibles técnicas que permitan acelerar los algoritmos existentes, reduciendo su tiempo de ejecución.

Pasamos ahora a describir con un poco más de detalle las ecuaciones de Navier-Stokes, la metodología de discretización del problema, y los tipos de mallas que pueden existir.

2.1.1 Ecuaciones de Navier-Stokes

El objeto de los métodos CFD, como ya se ha mencionado, es la resolución de una serie de ecuaciones, que modelan matemáticamente el comportamiento de los fluidos en un medio. El conjunto de leyes fundamentales que gobiernan el comportamiento de un fluido puede sintetizarse mediante las ecuaciones de Navier-Stokes, que vienen dadas por la siguiente expresión:

$$\frac{d}{dt}\int_{\Omega} W\, d\Omega + \int_{\delta\Omega} (F_c - F_v)\, dS = \int_{\Omega} Q\, d\Omega \qquad (2.1)$$

$$W = [\rho, \rho u, \rho v, \rho w, \rho E] \qquad (2.2)$$

dónde W son las variables conservativas, F_c y F_v son los flujos convectivo y viscoso respectivamente, y Q es un término que concentra toda la información relativa al calentamiento volumétrico y los choques entre cuerpos (en adelante, este término se considerará nulo). La integral se realiza sobre un dominio de integración Ω, que es la región en la que se encuentra situado el fluido.

Un caso particular de estas ecuaciones son las ecuaciones de Euler, en las que los flujos viscosos se consideran cero. Este tipo de ecuaciones se utilizan en aquellos problemas en los que la viscosidad es muy baja, y vienen dadas por una expresión del siguiente tipo:

$$\frac{d}{dt}\int_{\Omega} W\, d\Omega + \int_{\delta\Omega} F_c\, dS = \int_{\Omega} Q\, d\Omega \qquad (2.3)$$

La notación utilizada aquí es la misma que se sigue en [6], donde puede encontrarse, además, una descripción más detallada de estas ecuaciones. Asímismo, en [7] puede encontrar una descripción alternativa.

Antes de continuar desarrollando la solución, vamos a tratar brevemente el sentido físico de las ecuaciones, así como de las variables que las componen.
Lo que nos muestran las ecuaciones es que la variación del total de las variables conservativas (la integral de las mismas) con respecto al tiempo es igual a la componente volumétrica interna del objeto, menos el flujo a través de la frontera del mismo. Con un término $Q=0$, tendríamos que la variación total en las variables es igual al flujo total de las mismas a través de la frontera.
Estas variables, contenidas en el vector W, tienen cinco componentes: una de energía, ρE, que representa la energía total de cada región; tres direccionales, ρu, ρv y ρw, que representan la velocidad espacial de las partículas de la región; y una de densidad, ρ. Estas cualidades se transfieren de una partícula a otra mediante el choque entre ellas durante el movimiento del fluido, y eso es lo que representa la ecuación (2.1).

2.1.2 Resolución numérica de las ecuaciones de Navier-Stokes

Las ecuaciones de Navier-Stokes, presentadas en la sección anterior, carecen de solución analítica conocida. Por este motivo, para que sea posible su utilización, es necesario resolverlas de manera numérica. En esta sección se presenta, de modo breve y relativamente tosco, el modo en que se realiza este cómputo.
El proceso de resolución de este tipo de ecuaciones conlleva una integración espacial seguida de una integración temporal. Si consideramos que el influjo de Q es cercano a cero, el problema descrito en la ecuación (2.1) queda reducido a:

$$\frac{d}{dt}\int_{\Omega} W\, d\Omega = -\int_{\delta\Omega} (F_c - F_v)\, dS \qquad (2.4)$$

de donde queremos extraer W. De este modo, si tomamos

$$f(t,\Omega)=\int_{\Omega} W\, d\Omega \quad \text{y} \quad g(t,d\Omega)=-\int_{\delta\Omega}(F_c-F_v)\,dS \tag{2.5}$$

tenemos la siguiente expresión:

$$\frac{d}{dt}f(t,\Omega)=g(t,d\Omega) \tag{2.6}$$

De este modo, conocidas f y g, basta integrar la ecuación obtenida con respecto a t, y tendremos la solución. Por supuesto, este procedimiento dista mucho de ser trivial (ya se comentó al comienzo de la sección que no existe solución analítica conocida para estas ecuaciones).
Sí que existen, sin embargo, métodos numéricos que permiten calcular los flujos entrantes en cada uno de los puntos del dominio, una vez este ha sido discretizado. En el caso discreto, la integral pasa a ser una suma, de tal modo que se tiene la siguiente expresión:

$$\frac{d}{dt}f(t,\Omega)=-\sum_{\delta\Omega}(F_c-F_v) \tag{2.7}$$

Así pues, tenemos un problema que se reduce a computar cada uno de los flujos, sumarlos, y luego integrar la expresión anterior mediante algún método de integración de EDOs, como puede ser un Runge-Kutta, un método lineal multipaso, o un método implícito. Algunos de estos métodos pueden encontrarse descritos en [14].

2.1.2.1 Métodos implícitos vs explícitos

Una de los principales debates a día de hoy en el mundo CFD es la conveniencia del uso de métodos implícitos o explícitos a la hora de realizar el paso temporal.
Un método implícito es un método de integración numérica que asume, de forma teórica que se conoce el valor del paso que queremos calcular, estimándolo mediante un paso sencillo. Este tipo de métodos suelen ser muy rápidos en cuanto a número de iteraciones, pero requieren un cómputo muy pesado en cada iteración (ya que para su cálculo es necesario invertir una matriz).
En contraposición a este tipo de métodos están los métodos explícitos, que utilizan los pasos ya calculados para obtener el siguiente paso temporal. Estos métodos no son tan costosos por iteración, pero tienen un orden de convergencia mucho más bajo (es decir, tardan muchas más iteraciones en converger).
En suma, los métodos implícitos son más eficientes, pero son también más complicados de implementar, debido a los problemas que causan a la hora de utilizar la memoria o aplicar optimizaciones de cualquier tipo.

2.1.2.2 Mallas estructuradas vs no estructuradas

Una cuestión relevante a la hora de tratar este tipo de problemas es el modo en que están organizados los datos. Al realizar una discretización de la malla, es posible hacerla tanto estructurada como no estructurada.
Una malla estructurada es aquella en la que las caras están organizadas mediante un

esquema fijo (por ejemplo, una malla que se parte en cuadrados del mismo lado, o en triángulos equiláteros del mismo lado), mientras que una no estructurada es una malla en la que las caras están organizadas por un esquema variable, en el que puede haber, incluso, figuras geométricas diferentes.
En principio, podría parecer que una malla estructurada es una opción mucho mejor que una no estructurada, ya que en la primera mucha de la información va implícita (por ejemplo, se podrían saber las coordenadas de un punto con solo saber su índice). Sin embargo, una malla estructurada tiene importantes desventajas, siendo la principal la siguiente: En los problemas CFD, es habitual que determinadas regiones (las inmediaciones de un objeto, por ejemplo) requieran un cómputo mucho más fino que otras. Entonces, si la malla es estructurada y el problema contiene alguna región crítica, será necesario hacer un mallado muy fino en todo el espacio, lo que provocará un exceso de cómputo en regiones en las que es innecesario, además de un gasto de memoria adicional que puede llegar a sobrepasar al que implicaría la malla no estructurada, pese a la información que va implícita en la estructurada. Además, el hecho de que la información vaya implícita es un problema en sí mismo, ya que implica cierto cómputo adicional, lo que puede no interesarnos (como va a ser el caso).
En la actualidad, el solver CFD de referencia en el mercado +utiliza mallas no estructuradas, lo que es un reflejo de la ineficiencia del otro tipo de malla.

2.2 Trabajo previo en aceleración de algoritmos CFD mediante HW

En esta sección realizamos un breve análisis sobre el trabajo ya realizado en el mundo de la aceleración hardware, aplicado al mundo de los CFDs. Este análisis será fundamental a la hora de escoger el problema concreto que vamos a intentar resolver, ya que queremos que el trabajo sea original, y que siente las bases de la tecnología utilizada.
A continuación, mencionamos algunas de las tecnologías y trabajos ya existentes.

2.2.1 Tecnología hardware en procesamiento de información

Dentro del mundo de la aceleración por hardware existen distintas tecnologías válidas, de las cuales las principales son las FPGAs y las GPUs.
Las GPUs (Graphical Processing Units), utilizadas convencionalmente para el procesamiento de gráficos en ordenador, poseen una gran capacidad de procesamiento, pudiendo alcanzar rendimientos muy elevados. Las FPGAs, por su parte, no aportan un rendimiento tan alto, pero permiten un nivel de personalización del diseño mucho más elevado, lo que hace que el desarrollador disponga de mayor control sobre el funcionamiento del mismo. En particular, a la hora de aplicar paradigmas de paralelismo, el grado (potencial) de paralelismo alcanzado por un diseño FPGA es mucho mayor que el alcanzado por un diseño GPU.
Las FPGAs tienen algunas otras ventajas con respecto a las GPUs, como por ejemplo la interfaz. Mientras que una GPU sólo puede accederse mediante un bus PCI-express, las FPGAs ofrecen otros muchos puertos de E/S, permitiendo mayor control también en este aspecto en concreto.
En última instancia, la tecnología ideal dependerá del problema que se quiera resolver, aunque a priori parece que, para problemas más paralelizables, la FPGA permitirá

implementar una solución más eficiente.
De forma adicional a estos dos tipos de elementos de aceleración, están los clusters. Aunque la aceleración en cluster no es exactamente hardware, se basa en principios parecidos (la ejecución en paralelo de distintas partes del proceso), y por eso la mencionamos aquí.

2.2.2 Trabajos previos

Algunos trabajos previos (como por ejemplo, el citado en [1]) aplican pipelining sobre un solver Euler 2D, utilizando Impulse C como herramienta de traducción del lenguaje de alto nivel a lenguaje software.
Otra referencia respecto de este tipo de solvers es la aportada por Brandvik y Pullan en [3]. En este trabajo, se plantea cómo acelerar un solver de Euler en dos y tres dimensiones mediante el uso de GPUs. Este trabajo, al igual que el anterior, es una solución particular para un problema específico, que viene a demostrar la utilidad de las GPUs como herramienta de aceleración.
Por último, en otro de los trabajos citados ([2], se presentan técnicas de precompilado que permiten la ejecución eficiente de este tipo de algoritmos en clusters. Este trabajo es una solución relativamente general que permite paralelizar varias aplicaciones CFD en clusters, pero cuya eficiencia está limitada a la capacidad de procesamiento paralelo de este tipo de elementos de computación, muy inferior a la capacidad de las FPGAs.

Estos resultados, marginales, muestran que la aceleración por hardware de métodos CFD tiene ahora mismo un gran potencial. Además, el crecimiento de las herramientas de síntesis de alto nivel (como AutoESL o su versión más reciente, Vivado HLS) promete, a priori, facilitar considerablemente la tarea de adaptar estos algoritmos a lenguaje hardware.

3. Problema a resolver

Tras realizar el análisis del estado del arte en el área en la que se quiere desarrollar el trabajo, ya estamos listos para elegir el problema que queremos resolver, así como para plantear una solución conceptual para el mismo.
Este paso forma parte de una etapa previa a la implementación, en la que no se pretende describir la implementación de la solución (lo que se hará en las secciones 5 y 6), sino definir el problema que queremos tratar de forma teórica, esbozando el diseño que se adoptará más adelante.

3.1 Planteamiento del problema

El problema que hemos optado por resolver consiste en la aceleración de un solver CFD industrial, mediante técnicas de paralelización (pipeline) a nivel hardware. Este tipo de técnicas consisten en la ejecución simultánea de operaciones (en vez de ejecutarlas de forma secuencial, como sucede habitualmente cuando se ejecuta un programa), lo que permite reducir considerablemente el tiempo de ejecución del programa.
Esta ejecución simultánea puede realizarse únicamente si no existen dependencias de datos entre las distintas instrucciones. En general, existen tres tipos de dependencias de datos, que se explican brevemente aquí.
Read After Write (leer tras escribir): Una instrucción tiene que leer un registro, pero otra instrucción, adelantada durante el pipelining, escribe en dicho registro antes de que se complete la lectura.
Write After Read (escribir después de leer): Una instrucción tiene que leer el resultado de otra, pero, al paralelizar, lee la posición en la que la segunda instrucción iba a escribir antes de que esta se complete.
Write After Write (escribir después de escribir): Una instrucción escribe un resultado en una posición de memoria, y otra vuelve a escribir en la misma posición, antes de que una tercera tenga tiempo de leer el resultado que escribió la primera.

La tecnología sobre la que correrá este nuevo solver será una FPGA, dispositivo con gran capacidad de ejecución paralela de operaciones, que nos permitirá aprovechar al máximo este tipo de técnicas.
La elección de este planteamiento viene motivada por el análisis previo realizado en la sección 2.2, donde comentamos que las FPGAs son la tecnología que mayor grado de control nos proporciona sobre el diseño, permitiendo alcanzar un mayor grado de paralelismo que otras tecnologías.

Este proceso requerirá la traducción del solver a un lenguaje hardware (en este caso Verilog), sobre el que se podrán aplicar las mencionadas técnicas de pipeline con el objetivo de mejorar el tiempo de respuesta.
Nuestro objetivo de partida era lograr el máximo grado de aceleración posible, obteniendo, como mínimo, un 100% de aceleración (aunque un objetivo real de aplicación industrial debería lograr mucho más, en torno al 2000%), logrando así un demostrador de este tipo de tecnologías.

3.2 Dificultades

La resolución del problema descrito en la sección anterior plantea una serie de dificultades, que se describen a continuación:

En primer lugar, los solvers CFD industriales que nos gustaría adaptar operan sobre mallas. Estas mallas son un conjunto de datos relativos al problema termodinámico que va a resolver el solver, y contienen información sobre los puntos del espacio (coordenadas, identificadores), la vecindad de esos puntos, y sus variables termodinámicas.
Debido a la gran cantidad de información que contienen estas mallas, estas se encuentran almacenadas en ficheros externos al programa, lo que implica que al ejecutarse uno de estos solvers es necesario leer un fichero externo que contiene dicha malla, cargándola en memoria. Esta operación de lectura de ficheros no puede realizarse dentro de la FPGA, ya que las mallas están en la memoria del ordenador, a la que la FPGA no tiene acceso. Esta dificultad obliga a que la lectura sea realizada previamente por un proceso ubicado en el ordenador, que deberá enviar los datos a la FPGA.

Además, el tamaño de las mallas plantea una segunda dificultad, y es que rebasan con creces la capacidad local de la FPGA. Mientras que una malla de prueba sencilla (5000 puntos y 3000 aristas) ocupa X MBytes, la FPGA modelo Virtex-7 utilizada (una de las más avanzadas del mercado) tiene una capacidad de Y MBytes. En cuanto el tamaño de la malla crece un poco (unos Z puntos), las block-RAMs de la FPGA son insuficientes para abarcar la malla, y nosotros queremos que el solver sea capaz de abordar problemas industriales, cuyas mallas pueden llegar a alcanzar los 10 millones de puntos (lo que está muy por encima del valor soportado por la FPGA).
Esta cuestión es tan determinante que podría, por si misma, descartar las FPGAs como método de aceleración de este tipo de algoritmos. La solución a esta cuestión reside en el uso de una memoria DDR externa, que muchas FPGAs de última generación traen incorporada.

La tercera dificultad que plantea este tipo de solución es que la FPGA procesa a una frecuencia considerablemente más baja que un ordenador (100 MHz frente a los 3600 MHz que puede alcanzar un procesador de última generación). Esto implica que es necesario lograr un grado de paralelización muy alto si se quiere observar algún resultado.

3.3 Solución planteada: diseño heterogéneo

Para la implementación de un solver que responda a los objetivos presentados y sea capaz de sortear las dificultades mencionadas, se ha optado por el planteamiento siguiente: el solver a implementar será un sistema heterogéneo, en el sentido de que una parte del mismo se ejecutará dentro de un procesador (la encargada de la lectura de los ficheros de mallas, y de la escritura de los ficheros que contendrán las soluciones arrojadas por el solver), y la otra dentro de una FPGA (la encargada de realizar el procesamiento de los datos). La parte del solver que se ejecute dentro de la FPGA estará paralelizada mediante las técnicas de pipeline descritas en la sección 3.1, de tal modo que el tiempo de procesamiento pueda reducirse de forma considerable. La comunicación

entre ambos bloques se llevará a cabo a través del bus PCI-express del ordenador, mediante los protocolos que sean necesarios para ello.
Mediante esta solución, se consigue evitar uno de los principales problemas planteados en la sección anterior: la necesidad de cargar las mallas desde ficheros de datos presentes en la CPU. También se pretende, mediante la aplicación de técnicas de pipeline, compensar con creces la deceleración que supone bajar el solver a la FPGA.
Este diseño ha motivado la patente tecnológica referenciada en [13], a día de hoy aceptada para revisión.
El diseño se completa con la integración de la memoria DDR3 de la FPGA dentro del mismo. El uso de dicha memoria tiene como objetivo solventar el problema del tamaño de las mallas planteado en la sección 3.2, permitiendo un acceso rápido a la información de la malla que se está procesando por el segmento de código que se encuentra dentro de la FPGA.
En cuanto a la traducción de la parte del código C a código hardware, y la paralelización del mismo, será llevada a cabo utilizando la herramienta de síntesis de alto nivel Vivado HLS, herramienta desarrollada por Xilinx, que ofrece un entorno de programación cómodo para este tipo de traducción, disponiendo, además, de un conjunto de directivas que permiten implementar de forma automática y sencilla un gran número de optimizaciones hardware, y también generar una amplia variedad de interfaces.
En la sección 4 se describe con detalle la solución implementada, que se basa en la solución conceptual que se acaba de presentar.

4. Metodología de diseño con Vivado HLS

Antes de empezar a detallar el modo en que se implementa la solución descrita en la sección anterior es necesario explicar el modo en que se lleva a cabo el diseño con la herramienta Vivado HLS, explicación que se dará en los siguientes apartados.
Fundamentalmente, el objetivo de esta sección es explicar el modo en que se realiza un diseño con la herramienta mencionada, aprovechando de paso para explicar algunas de las cuestiones más críticas, como las opciones de generación de interfaces proporcionadas, dando algunos detalles sobre dichas interfaces.

4.1 Diseño con Vivado HLS

La herramienta Vivado HLS nos permite generar diseños hardware que repliquen la funcionalidad de un código software que le suministremos, siempre que esté escrito en C o en C++. Es decir, a partir de un código fuente escrito en C o C++ (que puede estar subdividido en varios ficheros), la herramienta generará de forma automática un código escrito en VHDL o Verilog cuyo comportamento será idéntico al del código C original.
Como podrá imaginarse, existen ciertas restricciones en lo relativo al código que se le suministre a la herramienta, las cuales se mencionan a continuación.

Cabecera del diseño
El diseño que le suministremos a la herramienta debe tener un top level, que no puede ser en ningún caso el main del programa. Esto nos exigirá, a menudo, la incorporación de un fichero en C que actúe como *top-level* del diseño.

Uso de punteros
El uso de punteros está bastante restringido en esta herramienta, que nos permitirá, como máximo, un nivel de indirección (es decir, podremos utilizar punteros simples pero no dobles).

Memoria dinámica
El uso de memoria dinámica está completamente prohibido en este tipo de diseños. Al ir a traducirse en un diseño hardware (con sus registros de memoria internos correspondientes), el tamaño del diseño debe estar perfectamente definido de antemano.

Entradas y salidas del programa
Las entradas y salidas del programa deben realizarse todas por los argumentos de la cabecera, no pudiendo leerse variables de ningún fichero, ni tampoco por la línea de comandos o cualquier otra vía. Obviamente, esto implica que no pueden realizarse operaciones como pintar una salida por pantalla.

Estas restricciones nos obligarán, generalmente, a realizar ciertas modificaciones en el código antes de sintetizar su versión hardware.

4.2 Simulación con Vivado HLS

Tras seguir los pasos detallados en la sección anterior, ya tendríamos nuestro diseño listo para ser sintetizado. Sin embargo, si hemos realizado modificaciones en el código (cosa bastante probable), podremos querer comprobar que el diseño sigue funcionando igual que al principio. Para este propósito, Vivado HLS nos brinda la posibilidad de realizar un "testbench" a nivel software.
Para realizar este testbench, habremos de incorporar a una subcarpeta llamada de ese modo una serie de funciones por encima del top-level hardware, que contendrán toda la lectura de argumentos, ficheros, y demás elementos de entrada y salida de datos. Lo normal es que nuestro antiguo main se incorpore en este paso, posiblemente con alguna modificación. En algún punto, nuestras funciones llamarán al top-level del diseño hardware, pasándole como argumentos todos los datos que necesite, así como las variables apropiadas para recibir de vuelta los datos que arroje éste.
Una vez hecho todo esto, estaremos listos para ejecutar a nivel software el diseño, comprobando que funciona adecuadamente. La interfaz del Vivado HLS, muy similar a la de Eclipse, nos proporcionará todas las herramientas necesarias para pasarle los argumentos al programa, ver la salida, e incluso depurar.

4.3 Interfaces del diseño

En este punto tratamos una de las cuestiones más determinantes en cuanto al diseño con esta herramienta, que es la generación de las interfaces del módulo. Como ya hemos comentado, Vivado HLS genera un código hardware (Verilog o VHDL) cuyo comportamiento es análogo al de nuestro código C. Sin embargo, ese código estará compuesto por una serie de módulos hardware, todos ellos integrados dentro de un módulo general (cuyo nombre será *_top, donde * es el nombre del top-level del diseño del Vivado HLS).
Ahora bien, ¿cómo se comunica este módulo con el exterior? Como todo módulo hardware, debe tener una serie de canales de entrada y salida por las que realizar esta comunicación, los cuales se corresponden con los argumentos del top-level de nuestro diseño (como se indicó en la sección 4.1). Además, existe la posibilidad de añadir un canal de comunicación a través del return, como se verá un poco más adelante.

Además de la cuestión relativa a cómo se generan las interfaces del diseño, hay otra: ¿Qué tipo de interfaces tendrá nuestro diseño?
Uno podría pensar que la herramienta se limitará a generar una serie de buses de datos capaces de contener los tipos de datos que le estemos pasando como argumento (generando, por ejemplo, un bus de 32 bits si le damos un argumento entero). Esto será así siempre que no le indiquemos lo contrario a la herramienta. Si queremos generar un tipo de interfaz diferente, podremos hacerlo añadiendo unas directivas especiales en el código (o en un fichero de directivas asociado al proyecto). En el manual de la herramienta se indican las directivas concretas necesarias para generar cada interfaz de las múltiples ofrecidas por la misma. Aquí se explicará únicamente el modo en que se generan las interfaces que nos harán falta a nosotros.

Antes de continuar, vamos a mencionar brevemente dos protocolos de comunicación que nos serán muy útiles más adelante: AXI y AXI-Stream. Estos protocolos serán descritos en

mayor profundidad en las subsecciones siguientes, explicando los principales puertos de control que se necesitan, así como el funcionamiento de los mismos.
Los protocolos AXI y AXI-Stream son, como ya hemos dicho, protocolos de comunicación. Estos protocolos funcionan a nivel hardware, permitiendo a los dispositivos que los implementen comunicarse entre sí de una forma robusta, rápida y estándar.
¿Para qué se usa cada uno de estos protocolos?
Principalmente, el protocolo AXI permite el envío y recepción de datos en ráfagas (es decir, el envío de grandes bloques de datos que pueden ser leídos en paralelo), mientras que el protocolo AXI-Stream permite el envío de datos mediante un modelo de streaming (es decir, los datos son enviados como una cadena ordenada, debiendo ser accedidos en el mismo orden en que han sido leídos). Un aspecto que diferencia ambos protocolos es (además de que son adecuados para modelos distintos de comunicación) que AXI es un protocolo bidireccional de comunicación (esto es, que un dispositivo puede tanto escribir como leer datos por una misma interfaz AXI), mientras que AXI-Stream es unidireccional, pudiendo únicamente enviar o recibir datos por una interfaz de este tipo (en función de si es maestra o esclava).

A continuación, se describen con detalle ambos protocolos, así como el modo de generar, mediante Vivado HLS, las interfaces necesarias para que nuestro diseño pueda comunicarse mediante este tipo de protocolos. También se explicará la generación de un tercer tipo de interfaz, que nos proporcionará una serie de puertos de control para nuestro diseño que serán útiles más adelante.

4.3.1 Protocolo AXI

Como ya adelantamos en la sección anterior, AXI es un protocolo de comunicación en ráfagas desarrollado por AMBA, muy útil a la hora de realizar transacciones de datos en las que estos se transmitan en paralelo. Este tipo de protocolo requiere una interfaz específica para poder ser llevado a cabo, además de seguir unos pasos concretos a la hora de llevar a cabo la comunicación.
Existen diversas versiones e implementaciones de AXI. Aquí describiremos la interfaz y protocolo necesarios para AXI4, en su implementación más restringida. Puede encontrarse una descripción mucho más detallada en [4].

La comunicación mediante el protocolo AXI se realiza a través de cinco canales independientes. Cada uno de estos canales dispone, además, de tres señales especiales, *valid, ready y last*, una de cada por canal, que permiten gestionar las comunicaciones (esto se verá un poco más adelante).
Es importante señalar también el hecho de que, en el protocolo AXI, uno de los extremos actúa como maestro (el que va a enviar los datos) y el otro como esclavo (el que los va a recibir).
Pasamos ahora a describir los canales de comunicación.

Canal de escritura
El canal de escritura se encarga de transportar los datos desde el maestro hasta el esclavo, admitiendo anchos de bus de 8 a 1024 bits de ancho.

Canal de lectura
Este canal transporta información desde el esclavo al maestro, siempre que este solicite una lectura de datos. El ancho de datos puede ser de entre 8 y 1024 bits.

Canal de dirección de escritura
Este canal transporta la información sobre la dirección en la que se van a escribir los datos durante una transacción de escritura de datos.

Canal de dirección de lectura
Este canal transporta la información sobre la dirección de la que se van leer los datos durante una transacción de lectura de datos.

Canal de respuesta a escritura (Write Response Channel)
Este canal transporta el reconocimiento (acknowledgement) del esclavo al maestro una vez se ha completado la escritura. Su uso no es obligatorio.

Puede encontrarse un esquema breve de la arquitectura de estos canales en las figuras 1 y 2.

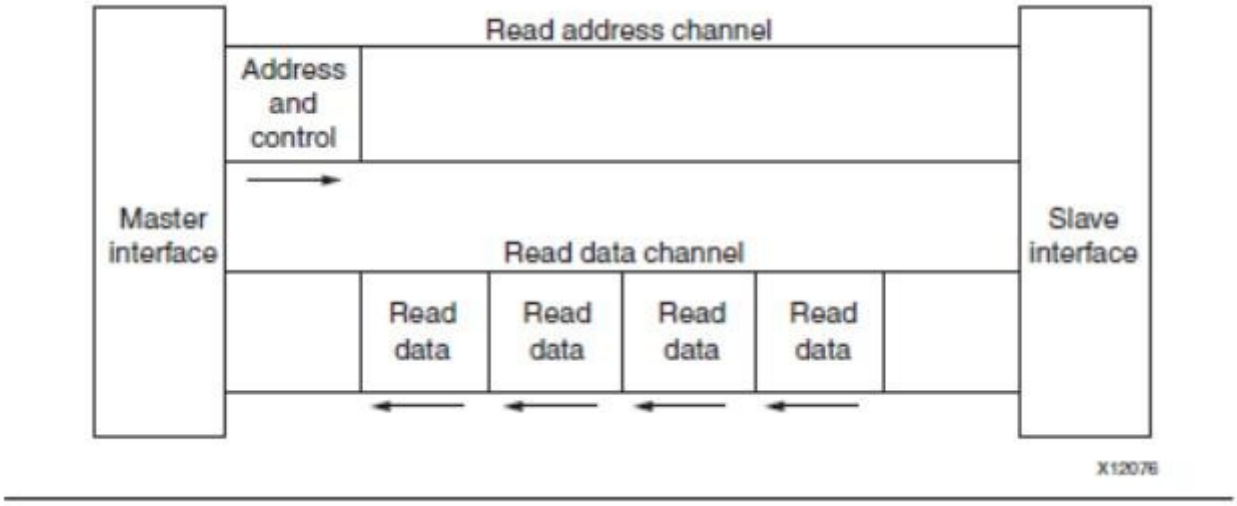

Figura 1 - Arquitectura del canal de lectura de AXI. Imagen obtenida de la especificación del protocolo AXI ([4]).

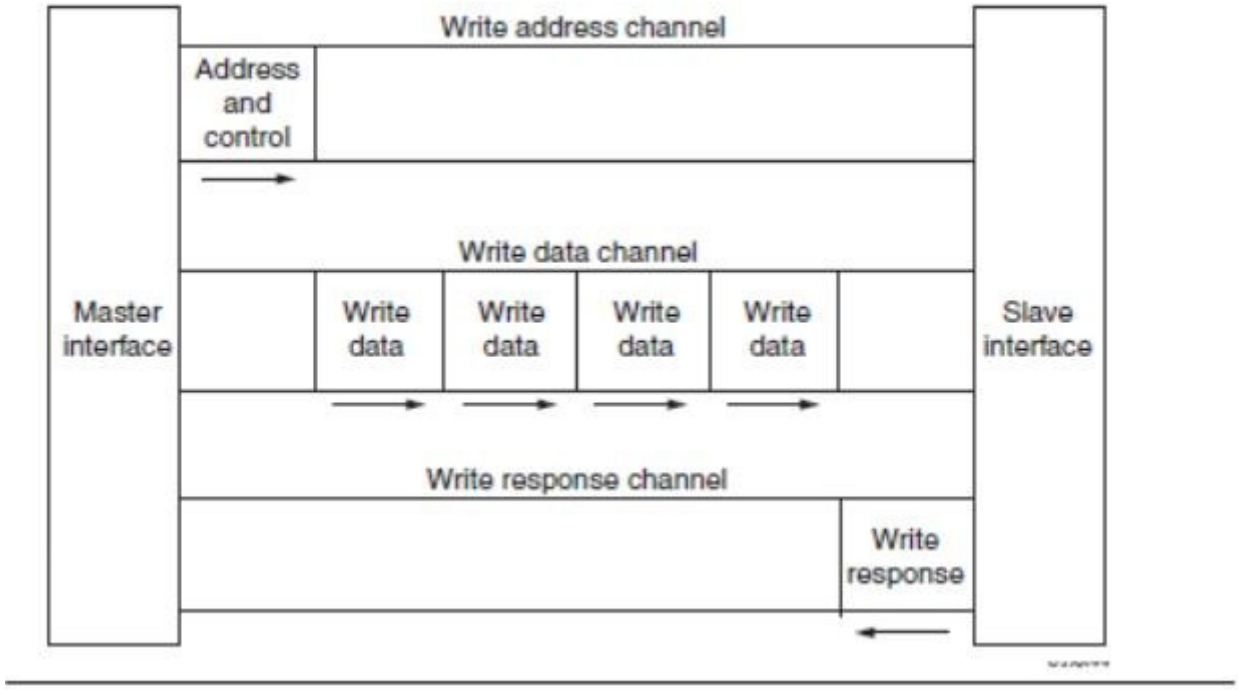

Figura 2 - Arquitectura del canal de escritura de AXI. Imagen obtenida de la especificación del protocolo AXI ([4]).

Una vez descritos los canales, pasamos a describir el procedimiento por el cual se realiza la comunicación a través de los mismos. Como ya se mencionó, esta comunicación sigue un protocolo que hace uso de tres señales especiales, denominadas VALID, READY y LAST. Cada canal dispone de sus propias señales de este tipo (de modo que, por ejemplo, las señales de control del canal de escritura serán WVALID, WREADY y WLAST). El protocolo de comunicación (handshake) entre maestro y esclavo también es el mismo para todos los canales, y es el siguiente:

El extremo que va a enviar los datos cargará los mismos en el bus adecuado, y, una vez hecho esto, pondrá la señal VALID a 1, indicando así al otro extremo que los datos están listos. Por su parte, el extremo que recibe los datos debe levantar la señal READY para indicar que está listo para realizar la transacción. El envío de datos a través de cualquiera de los canales se produce cuando ambas señales están en alto en un mismo ciclo de reloj. El orden en el que se levantan las señales carece por completo de importancia. La transacción puede darse, por tanto, en tres situaciones: si se levanta VALID y mientras aún está en alto, se levanta READY; si se levanta READY y, mientras aún está en alto, se levanta VALID, o si ambas señales se levantan simultáneamente. Estas tres situaciones están representadas en las figuras 3, 4 y 5.

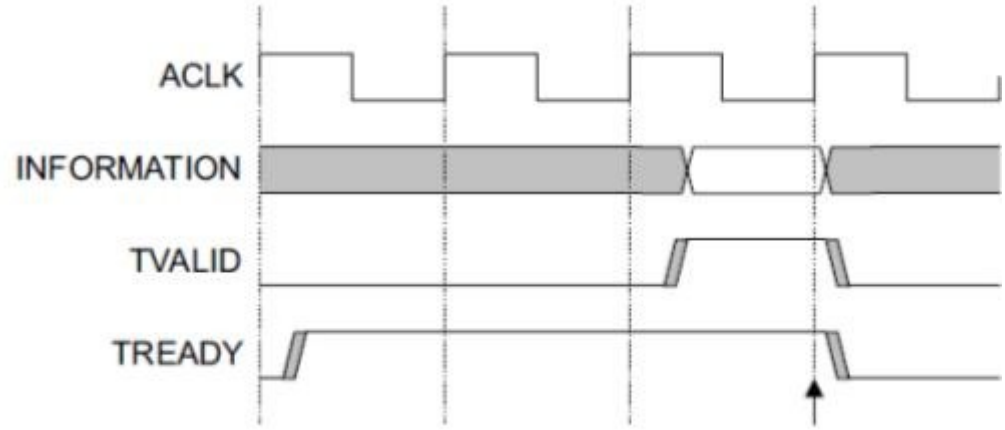

Figura 3 - Handshake AXI con levantamiento de READY previo al de VALID. Imagen obtenida de la especificación del protocolo AXI ([4]).

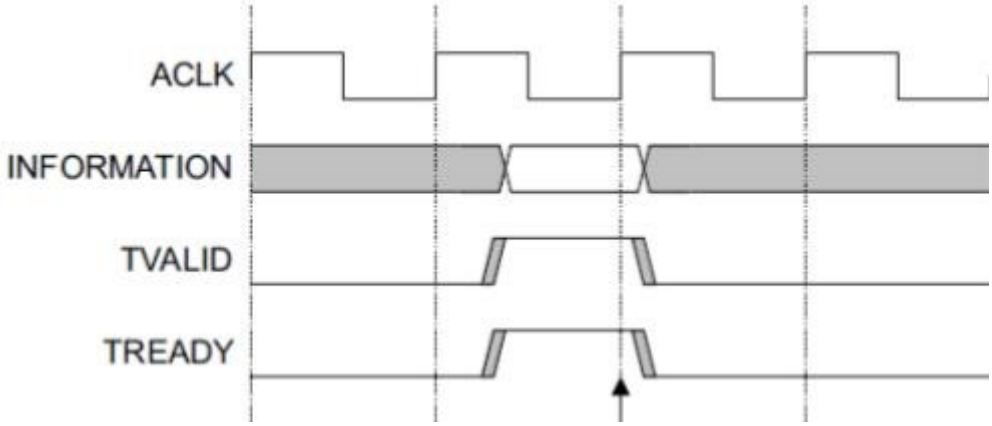

Figura 4 - Handshake AXI con levantamiento de READY y VALID simultáneos. Imagen obtenida de la especificación del protocolo AXI ([4]).

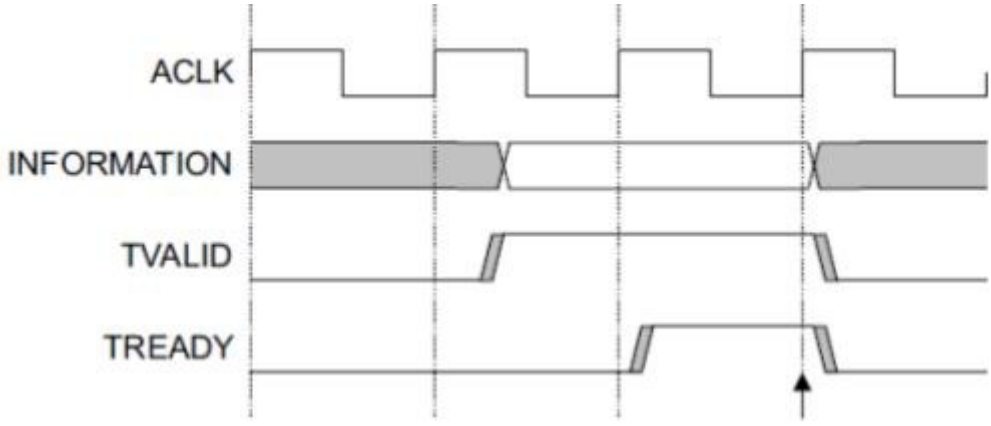

Figura 5 - Handshake AXI con levantamiento de VALID previo al de READY. Imagen obtenida de la especificación del protocolo AXI ([4]).

El único aspecto que queda por explicar es el uso de la señal LAST. Esta señal, concebida para indicar el final de una comunicación, permanecerá a cero en todo

momento durante la transacción, levantándose únicamente cuando el último bloque de datos esté llegando hasta el receptor. La señal se levantará a la vez que se levanta el último VALID, de modo que el receptor sabrá, al leer ese paquete de datos, que no van a entrar más, pudiendo continuar así con su procesamiento interno (lo que típicamente derivará en que la señal de READY baje, al menos temporalmente).

4.3.1.1 Interfaz AXI con Vivado HLS

Para conseguir una interfaz AXI es necesario, en primer lugar, modificar la cabecera del top level de nuestro diseño para que la variable que va a representar esta interfaz aparezca como argumento. En este caso, queremos un vector de datos sobre el que escribir y leer, al que llamaremos DDR_Stream. Así, la cabecera quedará del siguiente modo:

```
void fpga_main (    DATA entrada[MAX_STREAM],
                    DATA salida[MAX_STREAM]
                    volatile fDouble *DDR_Stream)
```

El atributo volatile es obligatorio ponerlo siempre que se vaya a utilizar una interfaz más de una vez, y dado que nuestra intención es leer y escribir cada dominio de la malla una vez por iteración, vamos a acceder bastantes veces a esta interfaz.

Una vez hecho esto, tenemos que indicarle a la herramienta que la interfaz correspondiente es un bus, lo que podremos hacer utilizando el siguiente pragma:

```
#pragma AP interface ap_bus port=DDR_Stream
```

De este modo, la herramienta sabrá que nuestra interfaz es un bus de datos, lo que habilitará la lectura y escritura en ráfagas, opción que nos permitirá acceder a los datos a gran velocidad.
Por último, queremos indicarle a Vivado HLS que nuestra interfaz se adapta específicamente al protocolo AXI4, y en particular que es una interfaz maestra, lo que se consigue con el siguiente pragma:

```
#pragma AP resource core=AXI4M variable=DDR_Stream
```

Una vez hecho esto, la herramienta se encargará de generar los wrappers correspondientes para que nuestro diseño disponga de la interfaz AXI4M que le hemos pedido.

Para utilizar esta interfaz, basta con que leamos o escribamos datos en la variable DDR_Stream. Si queremos realizar lectura o escritura en ráfagas (cosa que nos interesará, teniendo en cuenta que acelera bastante), habremos de utilizar la función memcpy, escribiendo o leyendo sobre la variable DDR_Stream según nos interese.

4.3.2 Protocolo AXI-Stream

El protocolo AXI-Stream es un protocolo de comunicación basado en FIFOs, muy útil cuando se quieren enviar datos de forma secuencial de un dispositivo a otro. Al igual que sucedía con AXI, vamos a describir este protocolo de comunicación, junto con sus interfaces, debido a que nos será de bastante utilidad más adelante.

Este protocolo de comunicaciones envuelve a dos extremos, entre los cuales la comunicación es unidireccional (es decir, uno de ellos envía los datos y el otro los recibe), cosa diferente a lo que sucedía en el protocolo AXI. Para que la comunicación entre ambos extremos pueda producirse es necesario que ambos extremos dispongan de la interfaz adecuada, que se compone de las siguientes señales:

- Bus de datos
- Valid
- Ready
- Last

Como habrá podido observarse, estas señales son las mismas que se utilizaban en el protocolo AXI en cada uno de los canales de comunicación. Esta coincidencia no es casual, ya que el protocolo de comunicación en este caso funciona exactamente igual al descrito en la sección anterior. Por este motivo, este protocolo no se describirá aquí de nuevo.

4.3.2.1 Interfaz AXI4-Stream con Vivado HLS

Para generar una interfaz AXI4-Stream, en primer lugar tenemos que añadir una variable a la lista de argumentos del top level de nuestro diseño. En este caso, el top level es la función denominada fpga_main, y queremos dos interfaces AXI4-Stream, que vamos a denominar “entrada” y “salida”. Así, la cabecera de la función quedará del siguiente modo:

```
void fpga_main (DATA entrada[MAX_STREAM], DATA salida[MAX_STREAM])
```

donde MAX_STREAM es la longitud máxima de nuestro stream, y DATA es una estructura de la siguiente forma:

```
typedef struct {
        uint64_t data;
        bool last;
} DATA;
```

donde el campo “data” contiene la información que queremos transmitir (en un entero de 64 bits) y el campo “last” indica si la posición es el último dato del paquete.
Como puede observarse, cada stream es una cadena de datos, que en cada posición lleva únicamente un dato, y la información sobre si es final de paquete o no.

Una vez añadidos los argumentos a la cabecera, llega el momento de indicarle a la herramienta qué tipo de interfaz queremos que sea. Para poder implementar un AXI4-

Stream, es necesario, en primer lugar, indicarle a la herramienta que la interfaz será de tipo FIFO, lo que se consigue con el siguiente pragma:

```
#pragma AP interface ap_fifo resource=entrada
```

De este modo le indicamos que el argumento “entrada” es una fifo. Se hace lo propio con “salida”:

```
#pragma AP interface ap_fifo resource=salida
```

Ahora, Vivado HLS ya sabe que entrada y salida son interfaces de tipo FIFO. Esto implica que si intentamos acceder de forma no secuencial a alguno de estos argumentos, la síntesis va a fallar (por ejemplo, si intentamos acceder a entrada[5] de forma aleatoria en un punto cualquiera del código, fuera de un bucle que recorra entrada).
Una vez hecho esto, ya estamos en condiciones de indicarle a la herramienta que queremos que esta interfaz se adapte de forma específica al protocolo AXI4-Stream, lo que se consigue con la siguiente directiva:

```
#pragma AP resource core=AXI4-Stream variable=entrada metadata=”-bus_bundle AXI4Stream_S” port_map={ {entrada_data TDATA} {entrada_last TLAST} }
```

Nótese que con la última opción le estamos indicando que la interfaz es de tipo esclavo (la interfaz “entrada” servirá para recibir datos del exterior, y por lo tanto sera esclava). Si queremos una interfaz maestra, como es el caso de la interfaz “salida”, el pragma será el siguiente:

```
#pragma AP resource core=AXI4-Stream variable=salida metadata=”-bus_bundle AXI4Stream_M” port_map={ {salida_data TDATA} {salida_last TLAST} }
```

En ambos casos estamos indicando explícitamente la presencia de la señal last, que nos permite manejar cómodamente los paquetes.

Para utilizar estas interfaces desde nuestro diseño, habremos de escribir (en el caso de la salida) o leer (en el caso de la entrada) las variables correspondientes, siempre accediendo a ellas de forma secuencial.

4.3.3 Puertos de control

Vivado HLS nos da la opción de añadirle a nuestro diseño un conjunto de tres señales de control, que nos permiten manejar el diseño dentro de la FPGA, indicándole que inicie su funcionamiento, y sabiendo si continúa ocupado o ya ha terminado su ejecución.
Estos puertos, denominados START, IDLE y DONE, actúan junto con el diseño del siguiente modo:

- En tanto que START permanezca bajo (antes de iniciarse la ejecución) el diseño permanecerá latente. IDLE estará a 1 y DONE a 0.
- Cuando START permanezca en alto durante un flanco activo de reloj, el diseño iniciará su ejecución, bajando IDLE a 0. START debería ser retornado a 0.
- La ejecución continúa hasta terminar. En ese momento, DONE se pone a 1 durante

un flanco de reloj, volviendo a continuación a 0. Mientras, la señal IDLE se pone a 1, permaneciendo así hasta que llegue un nuevo START

Es importante notar que, mientras que la señaL de START es una entrada de nuestro diseño (y por tanto, se gobierna desde fuera), las señales de IDLE y DONE son salidas, y por lo tanto son gobernadas por nuestro diseño.

4.3.3.1 Generación de los puertos de control con Vivado HLS

La generación de los puertos de control no requiere de la declaración de ningún argumento adicional, a diferencia de lo que sucedía con las otras interfaces. Para que dichos puertos se generen, basta con añadir en el código el siguiente pragma:

```
#pragma HLS INTERFACE ap_ctrl_hs port=return
```

El manejo de los puertos de control lo gestiona el propio diseño, por lo que no debemos de preocuparnos de ello a este nivel.

4.4 Integrando el hardware generado en nuestro diseño

Una vez la versión hardware de nuestro código ha sido sintetizada, ya estamos preparados para integrarla con el resto de nuestro diseño. Para ello, basta con copiar los ficheros .v generados (o .vhd, en caso de que generemos el diseño en VHDL) a una carpeta, y cargarlos desde el programa que estemos utilizando para sintetizar el diseño (por ejemplo, Xilinx ISE o Vivado). Una vez cargados los ficheros, podremos ver que todos ellos cuelgan de un solo módulo, que normalmente se llamará *_top, donde * es el nombre del top-level del diseño del Vivado HLS.

Dicho módulo tendrá las interfaces que le hayamos indicado que genere. De este modo, podremos conectar el módulo generado por Vivado HLS al resto de elementos de nuestro diseño, simplemente añadiendo los cables adecuados, y conectándolos a las entradas y salidas del mismo.

Una vez conectado el diseño, no hay que hacer nada más. El buen funcionamiento de toda la plataforma dependerá ya de que el resto del diseño hardware (y de otros elementos que pueda haber conectados) sea correcto.

4.5 Optimización del diseño con Vivado HLS

Una vez nuestro diseño está listo, habiendo sido correctamente adaptado, integrado y probado, ya estamos listos para optimizarlo. En realidad, el proceso de optimización podría llevarse a cabo en cualquier momento del proceso, siempre que el código ya haya sido adecuadamente adaptado para su síntesis con Vivado HLS.

Dejando a un lado otro tipo de optimizaciones posibles, la herramienta que estamos utilizando nos ofrece la posibilidad de configurar aspectos concretos del código hardware que se va a sintetizar (un ejemplo serían las directivas de interfaz, descritas en la sección 4.3). Esto se logra mediante la inserción de una serie de directivas, que puede realizarse en el mismo código o en un fichero especial de directivas que genera la herramienta para

este propósito. Vivado HLS ofrece un amplísimo catálogo de directivas, con un uso de lo más variado. Aquí explicamos algunas de las que nos permitirán mejorar el rendimiento de nuestro código, que serán las que nos harán falta, considerando los objetivos del trabajo.

4.5.1 Pipelining de bucles

Esta técnica permite que las operaciones dentro de un bucle se ejecuten de forma paralela. Si se impone la directiva correspondiente, Vivado HLS intentará que en cada ciclo de reloj se inicie una nueva iteración del bucle, aunque las anteriores no se hayan completado. El número de ciclos necesario para que pueda iniciarse una nueva iteración se denomina initiation interval (intervalo de iniciación). La herramienta intentará que este valor sea 1, salvo que le indiquemos lo contrario.
Para que sea posible obtener un II de 1, no puede haber dependencias de ningún tipo entre los datos. Los posibles tipos de dependencias son Read After Write, Write After Read, y Write After Write. La herramienta Vivado HLS detecta estas dependencias automáticamente, aunque a menudo detecta dependencias que no son tales, aspecto que puede solucionarse mediante la directiva de dependencia, descrita más adelante.

Para insertar la directiva correspondiente en el código, debe insertarse bajo la sentencia “for” el siguiente pragma:

```
#pragma HLS PIPELINE
```

La directiva puede insertarse también en el fichero de directivas.

4.5.2 Inlinining de funciones

Esta opción tiene como efecto la eliminación de la función de la jerarquía del programa, insertando su código de forma implícita dentro del cuerpo de la función que la llama. Aunque por si misma esta opción no causa efecto alguno en el rendimiento, si se aplica en aquellas funciones que son invocadas en bucles que están siendo paralelizados mediante pipeline puede ayudar, en ocasiones, a mejorar el rendimiento del sistema, haciendo que la paralelización afecte también a las operaciones internas de la función.

Para insertar la directiva correspondiente en el código, debe insertarse bajo la cabecera de la función el siguiente pragma:

```
#pragma HLS INLINE
```

La directiva puede insertarse también en el fichero de directivas.

4.5.3 Partición de arrays

El particionado de arrays es una directiva que se aplica sobre un array de datos. Si se utiliza, la herramienta partirá dicho array en elementos más pequeños, independientes entre sí. Esta operación hará que el acceso a las diferentes zonas de memoria sea más lento, a cambio de permitir que los accesos sean independientes. Esta opción acelerará el

proceso de lectura si se quiere acceder de forma sistemática a un mismo array de forma consecutiva. Dado que en un mismo elemento de memoria se puede realizar únicamente una operación de lectura o escritura simultánea, particionar el array cuando se quiere acceder de forma consecutiva a distintas posiciones del mismo permitirá mejorar considerablemente el rendimiento.

Para insertar la directiva correspondiente en el código, debe insertarse bajo la declaración del array el siguiente pragma:

#pragma HLS ARRAY_PARTITION variable=nombre_variable [complete|block|cyclic] dim=N

donde nombre_variable indica la variable a particionar, complete|block|cyclic el tipo de particionado, y N la dimensión sobre la que queremos particionar.
La directiva puede insertarse también en el fichero de directivas.

4.5.4 Restricciones de dependencia

Una de las cuestiones que más penalizan a la hora de paralelizar los bucles es la presencia de falsas dependencias. Debido a la estructura de los datos (organizados en muchos casos por arrays), la herramienta tiende a interpretar que existen dependencias de datos que realmente no son tales, cuando se intenta acceder simultáneamente a distintos elementos de un mismo array (por ejemplo, si se escribe en la posición 1 y se lee en la posición 2). En estos casos, puede conseguirse una mejora de rendimiento mediante la directiva DEPENDENCE, que nos permite indicarle a la herramienta que una dependencia no existe (también podríamos decirle que dicha dependencia de datos existe, si lo necesitásemos).
La sentencia para esta directiva es la siguiente:

#pragma HLS DEPENDENCE [inter|intra] [RAW|WAR|WAW] [true|false]

donde inter|intra indica si la dependencia es dentro de la misma iteración o entre iteraciones, RAW|WAR|WAW indica el tipo de dependencia, y true|false si la dependencia se da o no.

Este tipo de directivas debe utilizarse con cuidado, ya que si se inserta en un caso en el que sí que exista dependencia, indicándole al programa que tal dependencia es falsa, puede producirse un solapamiento de datos que lleve a un mal funcionamiento del programa.

5. Solución implementada

En esta sección se explica detalladamente la implementación de la solución planteada para el problema escogido. Tanto la descripción del problema como de su solución pueden encontrarse en la sección 3, y servirán como referencia en todo momento.
Antes de comenzar con la descripción, nos interesa recordar que nuestro objetivo es acelerar un solver CFD industrial mediante tecnología FPGA, y que la solución escogida implica el uso de una plataforma heterogénea, compuesta tanto por una FPGA como por un procesador.

5.1 Solver CFD de partida

Tal y como se comentó previamente, nuestro objetivo es adaptar un solver CFD con el objetivo de acelerarlo. Sin embargo, en el mundo de los CFDs hay muchísimos solvers, con características muy diferentes (en la sección 2.1 se explican algunas de estas características). Así pues, ¿cuál deberíamos escoger como punto de partida? Esta decisión, junto con las razones que la han motivado, se describen en las secciones 5.1.1 y 5.1.2 respectivamente. Además, en la sección 5.1.3 se realizará un análisis del código utilizado como punto de partida, con el objeto de determinar la estructura del solver heterogéneo que queremos implementar.

5.1.1 Detalles acerca del solver

El solver escogido implementa la resolución de las ecuaciones de Navier-Stokes en tres dimensiones, utilizando un esquema centrado para la integración espacial, y un método de Euler para la integración temporal.
El esquema centrado es de primer orden, lo que implica que para computar el residuo en un punto sólo se necesita disponer de la información relativa a los vecinos directos de este, hecho que facilita la paralelización.

La solución numérica sobre la que vamos a trabajar se organiza en dos ciclos principales: un bucle externo (llamado outer loop) que controla el valor del residuo (una variable física relacionada con la convergencia del método) y un bucle interno (llamado inner loop) que pasa a través de todos los puntos de la malla y calcula los nuevos valores de las variables termodinámicas que nos interesan. Es en este bucle interior donde se invocan las funciones de cálculo: el cálculo del flujo, el paso del Runge-Kutta (que hace la integración temporal), el cálculo de limitadores, y el cálculo del tamaño de paso de tiempo.
A pesar de esta organización "física", conceptualmente, lo que estamos haciendo es una integración espacial en el bucle interior, y una integración temporal en el bucle externo. En la parte inferior, se describe con cierto detalle lo que hemos hecho en la aplicación de cada uno de los diferentes elementos del programa de solución.

Entramos ahora un poco más en detalle:
En primer lugar, hablamos de la integración temporal. Tras realizar la integración espacial, la temporal se reduce a una ecuación diferencial ordinaria en una sola variable (el tiempo), que resolveremos mediante un método de Euler.
Este método explícito es bastante sencillo, pero funcionará bien en nuestro demostrador (aunque de cara a futuras mejoras será necesario sustituirlo por un método más estable).

Describimos ahora con un poco más de detalle el método de Euler.
Para entender este método, tenemos que imaginar que queremos computar el valor de una función f dada en función del tiempo, esto es, f(t), y que conocemos su derivada (f'(t)) y su valor en un punto (típicamente, f(t=0)).
Dando un poco de notación:

$$f'(t)=F(t) \quad \text{y} \quad f(0)=y_0$$

Con esta notación, el método de Euler se escribe del siguiente modo:

$$y_{n+1}=y_n+dt\,F(t_n)$$

donde y_n es el valor estimado de f en el punto t_n, y dt es la longitud del paso de integración escogida, que puede ser fija o dinámica.
En nuestro caso, el método elegido implementa también una selección dinámica del paso de integración (esto es, la "longitud" temporal de cada paso), lo que contribuye a aumentar la estabilidad del mismo. Esta selección dinámica del paso se basa en el valor de las variables en cada punto (a mayor valor de las primvars, menor es la longitud del paso), siendo siempre cuidadosos con no tomar una longitud de paso demasiado larga.
Este proceso se lleva a cabo en todos los puntos de la malla, dependiendo cada punto sólo de su estado previo (es decir, cada punto se computa independientemente del resto), por lo que requerirá que se recorran todos los puntos de la malla.

Hablamos ahora de la integración espacial. Este procedimiento consiste en el cómputo del flujo de las cinco variables termodinámicas asociadas a cada punto (descritas en la sección 2.1), y requerirá del recorrido de toda la malla por aristas.
El procedimiento se describirá de forma breve, sin entrar en mucho detalle, ya que las fórmulas del cálculo de los flujos son bastante complejas, y no merece la pena detallarlas aquí (aunque puede encontrarse el cálculo en [6]). De forma general, el cómputo se hace del siguiente modo:
En primer lugar, se recorren las aristas de la malla, calculando el flujo que atraviesa cada una de ellas. El flujo es un vector de cinco puntos, que se corresponden con una de las cinco variables termodinámicas que se quieren estudiar, y se calcula utilizando los valores de esas variables en los puntos que hay a los lados de la arista. Según se van calculando los flujos, se van sumando las contribuciones de los mismos a los flujos de entrada de cada punto, de modo que cuando se han recorrido por completo las aristas, ya se saben los flujos que entran y salen en cada punto.
Una vez calculados los flujos, ya está completa la integración temporal. Estos flujos se corresponden con la función derivada con respecto a t que queremos integrar en el paso temporal, por lo que ya podemos aplicar el método de Euler, descrito previamente en esta misma sección.

Una última cuestión que queda por mencionar es el cálculo de los limitadores. Estos limitadores son unos parámetros que acotan los flujos, de tal modo que un flujo nunca pueda ser mayor que su limitador. Este tipo de técnica se utiliza para ayudar a la estabilidad del método, ya que los valores de flujo muy grandes pueden provocar oscilaciones que hagan diverger el algoritmo rápidamente. En nuestra versión, se han elegido limitadores de primer orden, que utilizan únicamente el valor de los vecinos

directos del punto para ser calculados.
Un aspecto importante acerca de la implementación que se va a utilizar como punto de partida es que esta es multidominio. Esto quiere decir que la malla sobre la que se va a trabajar estará dividida en varios dominios, que podrán ser procesados de forma independiente, siendo necesario duplicar la información relativa a las fronteras de dichos dominios, de modo que cada dominio pueda disponer de la información que necesita acerca de los dominios vecinos.
Este planteamiento es posible gracias a que el solver utilizado es explícito, ya que, de otro modo, no podría procesarse la malla por partes, al necesitarse información no estrictamente local.
Este aspecto del solver de partida es muy importante, ya que nos permite solventar uno de los problemas mencionados en la sección 3.2, que es el problema derivado del tamaño de las mallas. Gracias a que el solver es multidominio, podremos cargar cada dominio de la malla en la FPGA y procesarlo por separado, evitando así el tener que tener la malla completa cargada, algo imposible en cuanto esta disponga de un número moderadamente grande de puntos.

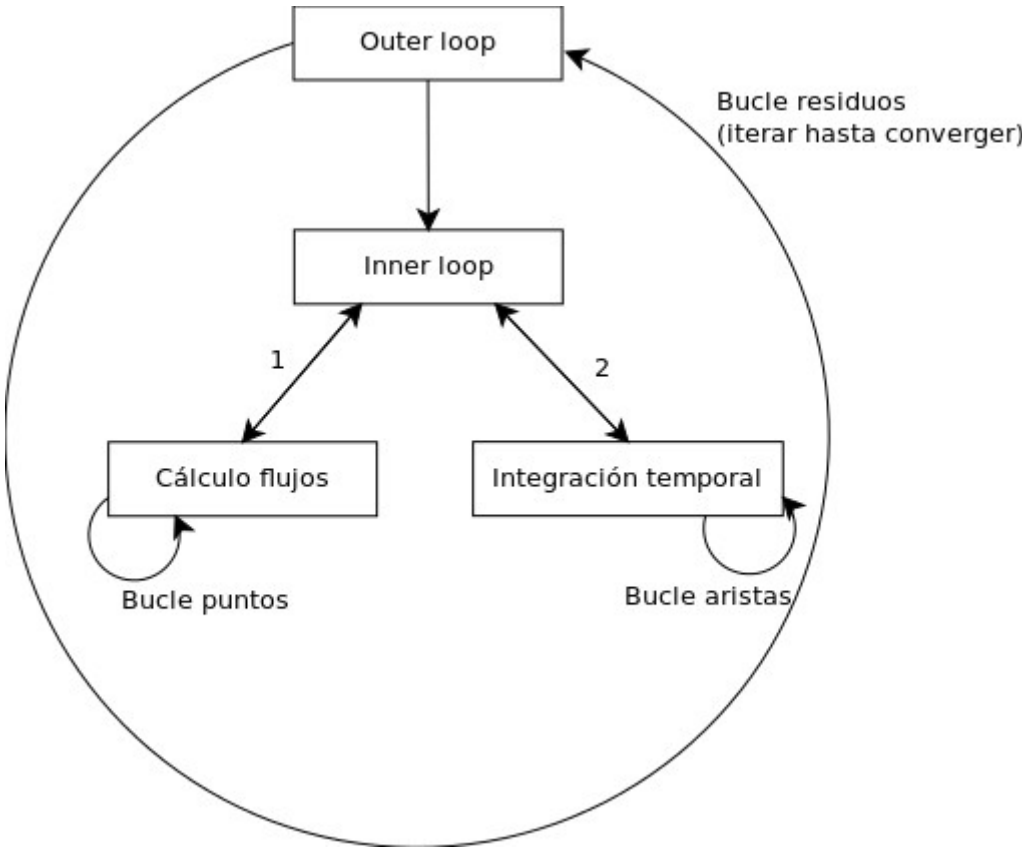

Figura 6 - Solver de partida. Los ciclos representan bucles, junto con las llamadas internas

5.1.2 Motivación para la elección del solver de partida

El solver elegido es relativamente sencillo, y muchos de los aspectos que implementa (el método de Euler para la integración temporal, el esquema centrado de primer orden para

la espacial) están muy superados en el mundo CFD. La elección de este solver, sin embargo, tiene un sentido, y es que el objetivo de este trabajo es demostrar la validez de la tecnología FPGA como herramienta de aceleración dentro del mundo CFD. Por este motivo hemos considerado que no valía la pena escoger un solver más complicado, cuya complejidad podría causar problemas de área dentro de la FPGA (a más complejidad, más operaciones, y por tanto mayor número de elementos internos utilizados), así como dificultar el proceso de paralelización al aumentar la dependencia entre los datos.

5.1.3 Análisis del solver de partida

En esta sección se presenta el análisis realizado sobre el código de partida, análisis necesario de cara a la toma de las decisiones de diseño posteriores. En particular, se presentará el análisis de la carga de esfuerzo realizada por el programa y el flujo de los datos a través del mismo. Esto nos ayudará a decidir qué partes del código se ejecutarán dentro de la FPGA y qué partes del mismo lo harán fuera, y también nos permitirá formarnos una idea de la infraestructura que será necesaria para hacer funcionar el diseño.

5.1.3.1 Profiling del código

El profiling del código es una técnica que nos permite ver el esquema de llamadas de las diferentes funciones que componen un programa. En nuestro caso, lo realizamos sobre el solver que utilizamos como punto de partida, para hacernos una idea de qué funciones lo componen, y de cuales de estas funciones se llevan mayor tiempo de ejecución. En la figura H puede observarse este esquema de llamadas, conteniendo las funciones más costosas y el flujo principal del programa.
Como se puede ver en dicha figura, la mayor parte del peso del programa recae sobre la función denominada outer_loop_md, función que se corresponde con la ejecución del bucle exterior. Esta función se lleva el 95% del tiempo de ejecución, aunque es importante tener claro que dentro de la misma se ejecuta también la función inner_loop, que realiza buena parte del procesamiento.
En cuanto al resto de funciones, muchas de ellas no son siquiera visibles (la figura no muestra las funciones que abarcan menos del 1% de tiempo de procesamiento, para facilitar el entendimiento del flujo de programa), y su función es, principalmente, leer las mallas, cargar la información en las estructuras de datos, y, tras completarse el procesamiento, escribir los resultados en ficheros externos.

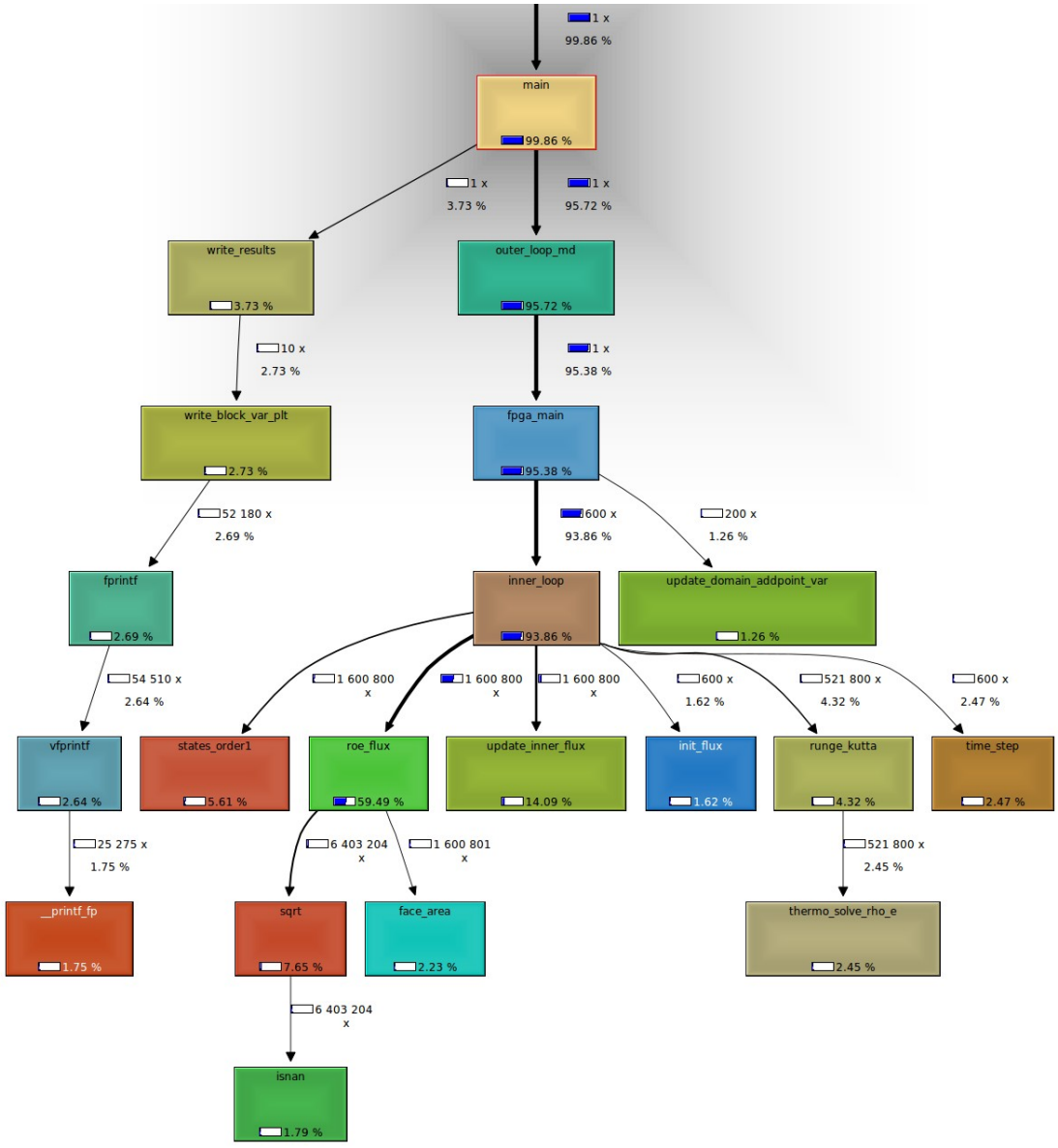

Figura 7 - Profiling del solver de partida

5.1.3.2 Análisis de los datos

El modelo que se está utilizando maneja una malla computacional en la que pueden distinguirse dos tipos fundamentales de elementos: los puntos y las aristas. Los puntos representan puntos del espacio en el que se está realizando la simulación, mientras que las aristas representan la conexión entre dichos puntos. La información contenida en el programa, por su parte, no se estructura estrictamente por estos objetos, haciendo uso de otro tipo de estructuras, que están, eso sí, asociadas a los mismos. Se presentan a continuación estas estructuras, junto con una breve explicación de las mismas:

facedata fdata[N_FACES]
Estos son los datos relativos a cada una de las aristas de la malla. Contiene el identificador de la arista, los puntos izquierdo y derecho, y algunos otros datos de interés.

fDouble pvolume[N_POINTS]
Contiene la información sobre el volumen del punto (variable Q de la ecuación 2.1)

fDouble primvars[N_POINTS][N_VAR]
Contiene la información sobre las variables termodinámicas del punto (variable W de la ecuación 2.1), siendo una matriz de N_POINTSxN_VAR, donde N_VAR es 5 (el número de variables termodinámicas).

fDouble flux[N_POINTS][N_FLUX]
Contiene los flujos que convergen a un punto .

boundarydata bdata[N_BOUNDARIES]
Contiene la información relativa a la frontera del dominio, cuyo procesamiento no se trata con detalle, por tener un coste computacional muy bajo en relación al resto (aunque, obviamente, debe realizarse).

En la sección siguiente se describe el modo en que se inicializan estos datos, así como la afluencia de los mismos a lo largo del programa.

5.1.3.3 Flujo de datos

Los datos del programa comienzan, con la salvedad de los flujos, escritos en un fichero de malla, que será leido por el programa principal invocando a las primitivas adecuadas.
Una vez leídas las mallas y cargados los datos en el fichero de entrada, el programa invoca a la función que realiza el bucle exterior (outer_loop_md), pasándole en el proceso la información leída de las mallas (toda la información relativa a los puntos y a las aristas, con la excepción de los flujos). Esta función, a su vez, enviará todos los datos a la función inner_loop, encargada del bucle interno.
Una vez dentro de inner loop, se invoca a la función init_flux, donde se inicializan los flujos a cero. A continuación, se invoca a la función states_orden1, que calcula los limitadores (aquí denominados states) a partir de las primvars. Tras esto, se llama a la función flux_roe, que calcula los flujos, pasándole los flujos de partida, las primvars, los states y los datos de las caras (fdata). Esta función retorna los flujos ya calculados.
Otra de las funciones invocadas es la función time_step, a la que se le pasan los primvar y los volúmenes (pvolume) para obtener el paso temporal (dtvol).
Con todos estos cómputos ya hechos, se invoca a la función runge_kutta, a la que se le pasan los flujos, los primvars, y el paso temporal (dtvol), y que retorna los primvar actualizados.

El flujo de datos puede verse representado en el esquema de la figura 8.

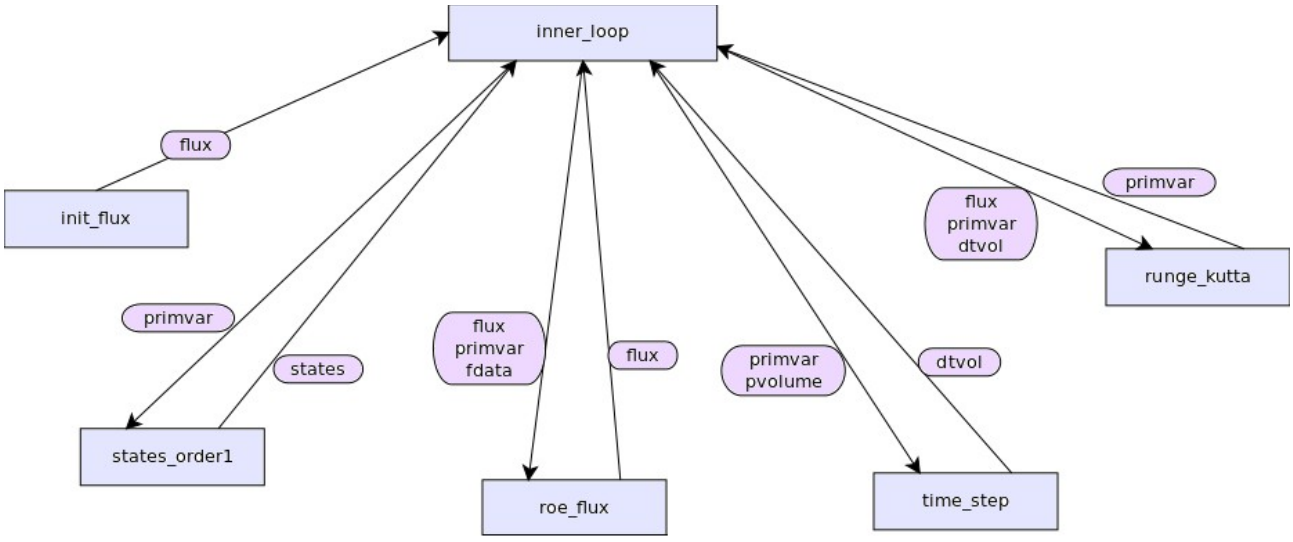

Figura 8 - Flujo de datos en la región interna del programa

5.2 Subdivisión del solver

Una vez planteado el solver sobre el que vamos a trabajar, nuestro objetivo es la construcción de un solver heterogéneo (como se explicó en la sección 3.3) a partir del mismo. Para ello, debemos dividir el solver del que partimos en dos bloques, uno de los cuales se ejecutará en el ordenador (el encargado de la lectura de las mallas de datos y de la escritura de las soluciones en ficheros), ejecutándose el otro en la FPGA (el encargado de la realización de todo el procesamiento). A continuación se describe el modo en que se ha realizado esta división, teniendo en cuenta lo explicado en la sección 4, en la que se habló de la metodología de diseño HLS.

5.2.1 Bloques software y hardware

Observando la figura 8 podemos ver que el solver de partida se divide en dos grandes regiones: una ocupada del procesamiento (todo lo que cuelga de outer_loop_md) y otra ocupada de la lectura y escritura de datos en los ficheros. Hay que tener en cuenta que, tal y como se planteó en la sección 3.2, la parte ocupada de la lectura y escritura de ficheros debe ejecutarse a nivel de procesador, mientras que la parte que se ocupa del procesamiento debería ejecutarse en la FPGA. Por ello, hemos dividido el solver en dos bloques conceptuales: el “bloque de lectura/escritura”, compuesto por todas las funciones que cuelgan del main y no pasan por el nodo outer_loop_md; y el “bloque de procesamiento”, compuesto por todas las funciones que cuelgan del nodo outer_loop_md, incluido él mismo. Estos bloques se denominarán también “bloque software” y “bloque hardware” respectivamente, ya que, como se ha comentado ya en varias ocasiones, el primero se ejecutará en el ordenador, mientras que el segundo se bajará a la FPGA, previo tratamiento para su optimización.

5.2.2 Adaptación con vivado HLS y optimización del rendimiento

Tal y como se ha explicado en las secciones anteriores, el bloque de procesamiento va instalado en la FPGA, lo que requiere de que sea traducido a lenguaje hardware, procedimiento que se ha llevado a cabo utilizando la herramienta Vivado HLS. En las subsecciones siguientes describimos cómo hemos aplicado este procedimiento al caso particular del problema que estamos resolviendo, detallando los aspectos modificados dentro del código y las interfaces que se han generado, así como las optimizaciones aplicadas al mismo.

5.2.2.1 Adaptación del código

En esta sección describimos brevemente el proceso de adaptación del código para su síntesis con la herramienta Vivado HLS. Dicha adaptación consiste en una serie de modificaciones que permitirán que el código cumpla con las restricciones detalladas en la sección 4, imprescindibles para que la herramienta sintetice con éxito el código hardware equivalente.
Las modificaciones realizadas son las siguientes:

1. Añadido de un módulo *top-level*:

La primera modificación a realizar es el añadido de un nuevo módulo dentro del código. Dicho módulo, denominado fpga_main, se sitúa entre el módulo llamado outer_loop_md del código original y la función inner_loop, siendo invocado por el primero e invocando al segundo. El flujo de llamadas del programa tras esta modificación puede verse en la figura 9.

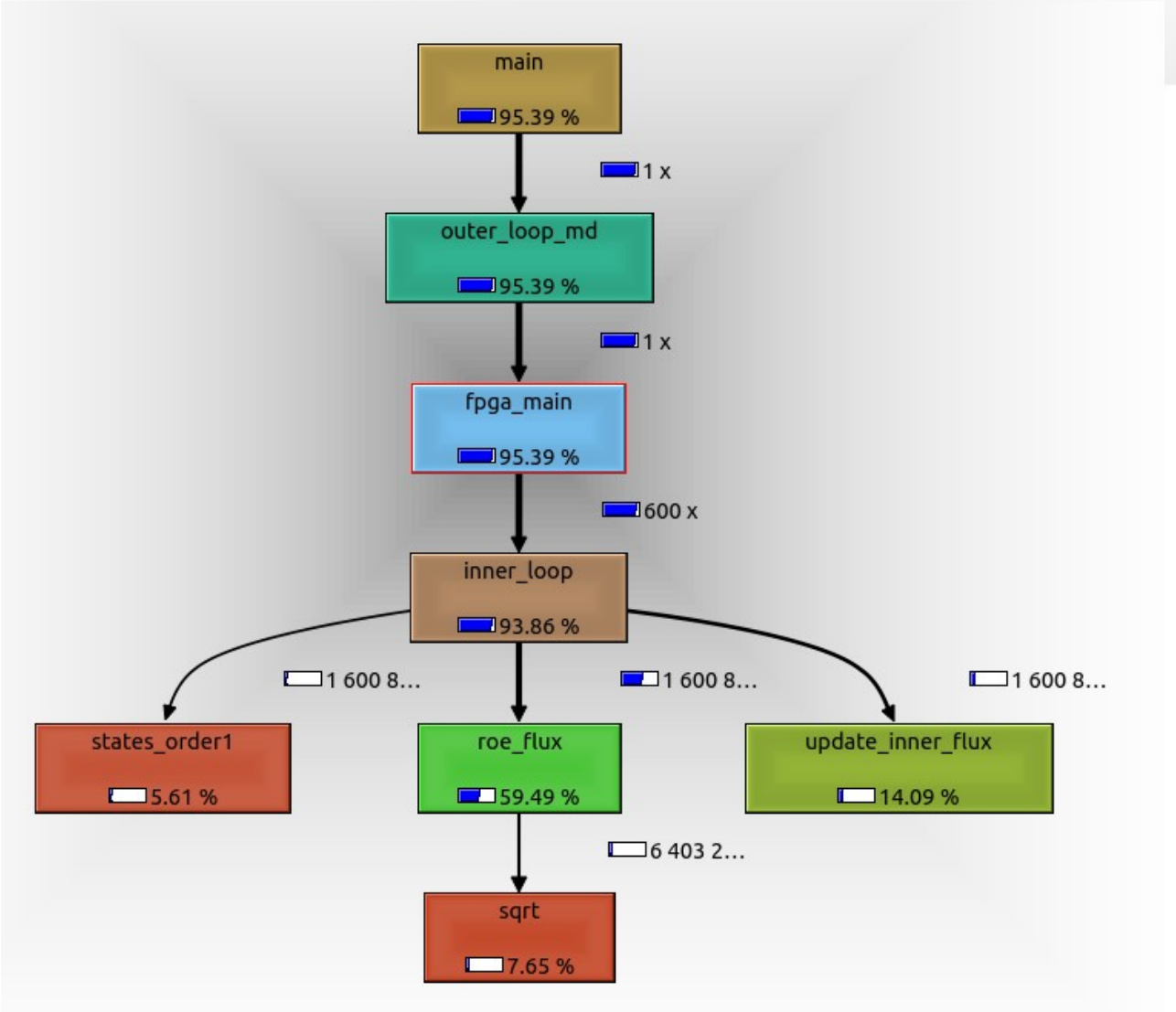

Figura 9 - Profiling del solver tras su adaptación (ejecutado al completo en software)

2. Creación de las estructuras estáticas necesarias

Todas las estructuras de datos mencionadas en la sección 5.1.3.2, son, al comienzo del programa, dinámicas (por comodidad a la hora de leer las mallas). Por otra parte en la sección 3 se explicó que el código que vaya a ser sintetizado por la herramienta no puede tener contenidos dinámicos, por lo que se deberán crear nuevas estructuras, estáticas, para contener los datos que inicialmente se encuentran en estructuras dinámicas.
Esta generación se produce en el cuerpo de la función llamada fpga_main (mencionada en la sección anterior), que recibe como argumentos todos los datos necesarios para la ejecución del programa.

3. Modificación de las cabeceras de las funciones.

Además de modificar las estructuras de datos para que sean estáticas, deben modificarse ahora las cabeceras de las funciones, que originalmente recibían punteros a estas estructuras, para que reciban ahora matrices y vectores de tamaño fijo.

5.2.2.2 Generación de las interfaces

Pasamos ahora a describir con detalle cómo generar cada una de las interfaces que debe tener nuestro diseño. Queremos que el diseño generado por Vivado HLS disponga de las siguientes interfaces:

- Dos interfaces AXI4-Stream.
- Una interfaz AXI
- Un conjunto de tres señales de control de actividad (start, idle, done).

El modo en que se generan estas interfaces utilizando la herramienta está descrito en la sección 4.3. El objetivo de estas interfaces es, principalmente, comunicar el módulo que generaremos con la herramienta con la memoria DDR3, y con el controlador del bus PCI-express, además de poder controlar, desde el nivel software del diseño, el estado de la ejecución de dicho módulo. Estos aspectos se describirán de forma detallada un poco más adelante.

5.2.2.3 Optimización del código utilizando directivas

En la sección 4.5 se describieron algunas de las directivas de optimización que pueden aplicarse al código que se va a sintetizar con la herramienta Vivado HLS. Estas directivas permiten mejorar ciertos aspectos del funcionamiento del código, en particular el rendimiento, que es la mejora en la que estamos interesados.
En el código se han introducido muchas directivas, pero principalmente directivas de pipeline en los bucles del mismo, con el objeto de alcanzar una importante mejora del rendimiento. Los resultados arrojados por la herramienta (en términos del rendimiento) se presentan en la sección 7.

6. Arquitectura de la solución

6.1 Entorno de trabajo

Antes de continuar explicando el modo en que se realiza la implementación del solver heterogéneo, es necesario perder un poco de tiempo en describir el entorno en el que vamos a trabajar, los dispositivos que vamos a utilizar, y cómo están conectados entre sí. Esto es algo imprescindible para entender las secciones siguientes, cuya existencia viene motivada por la necesidad de detallar el modo en que se maneja este entorno. Por eso, en esta sección presentamos tanto el entorno físico de trabajo, como un detalle acerca del soporte de toda la plataforma.

El entorno de trabajo consiste en un procesador Intel de 8 núcleos, conectado mediante un bus PCI-express a una FPGA modelo Virtex-7 (Xilinx). La FPGA dispone de una memoria DDR3 externa, que será utilizada como medio de almacenamiento. En la figura 10 puede encontrarse un esquema detallado del entorno de trabajo utilizado.

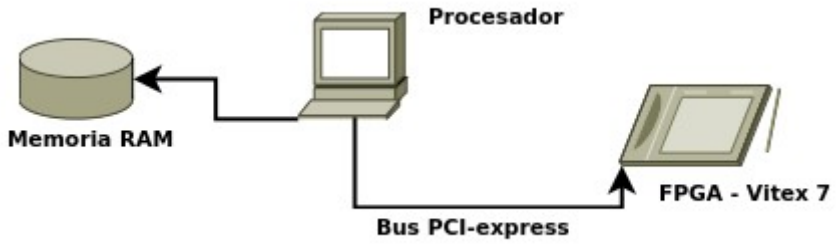

Figura 10 - Entorno de trabajo

La el modelo de tarjeta FPGA utilizada es Virtex-7 XC7VX485T-2FFG1761C . Dicha tarjeta contiene además los siguientes elementos adicionales:

- 1 GB DDR3 memory SODIMM
- 128 MB Linear byte peripheral interface (BPI) Flash memory
- USB 2.0 ULPI Transceiver
- Secure Digital (SD) connector
- USB JTAG through Digilent module

En cuanto al procesador utilizado, es un procesador de Intel con 8 núcleos, 8 GBytes de RAM, que funciona a 4 GHz.

6.2 Arquitectura general de la solución

Una vez descrito el entorno de trabajo, pasamos a detallar cómo estará organizado el sistema a nivel lógico. Como ya comentamos en la sección 5.2, nuestro código se divide en dos bloques, uno de los cuales se ejecutará en el ordenador,mientras que el otro lo hará en la FPGA. Dado que la ejecución de nuestro solver requiere de que ambos

bloques participen en el procesamiento, es necesario que puedan comunicarse entre sí, hecho que no es trivial considerando que uno de los bloques se encuentra en la FPGA y el otro en el ordenador. Como ya vimos en la sección anterior, las comunicaciones entre la FPGA y el ordenador se realizan a través del bus PCI-express, lo que implica que, para lograr la comunicación entre el bloque software y hardware, será necesario añadir a nuestro diseño los controladores adecuados, de tal modo que tanto el bloque software como el hardware sean capaces de enviar y recibir datos a través del bus PCI-express.
Otra cuestión de gran importancia a la hora de organizar la estructura del diseño es la necesidad de hacer uso de una memoria externa para almacenar las mallas, cuyo tamaño es muy superior a la capacidad local de la FPGA (tal y como se explicó en la sección 3.2).

A continuación se presenta un diagrama de bloques en el que puede verse explicada la arquitectura de nuestro diseño de forma general, sin entrar en detalles acerca de los elementos que componen cada parte del mismo, de los que hablaremos algo más adelante. Dicho diagrama puede encontrarse en la figura 11.

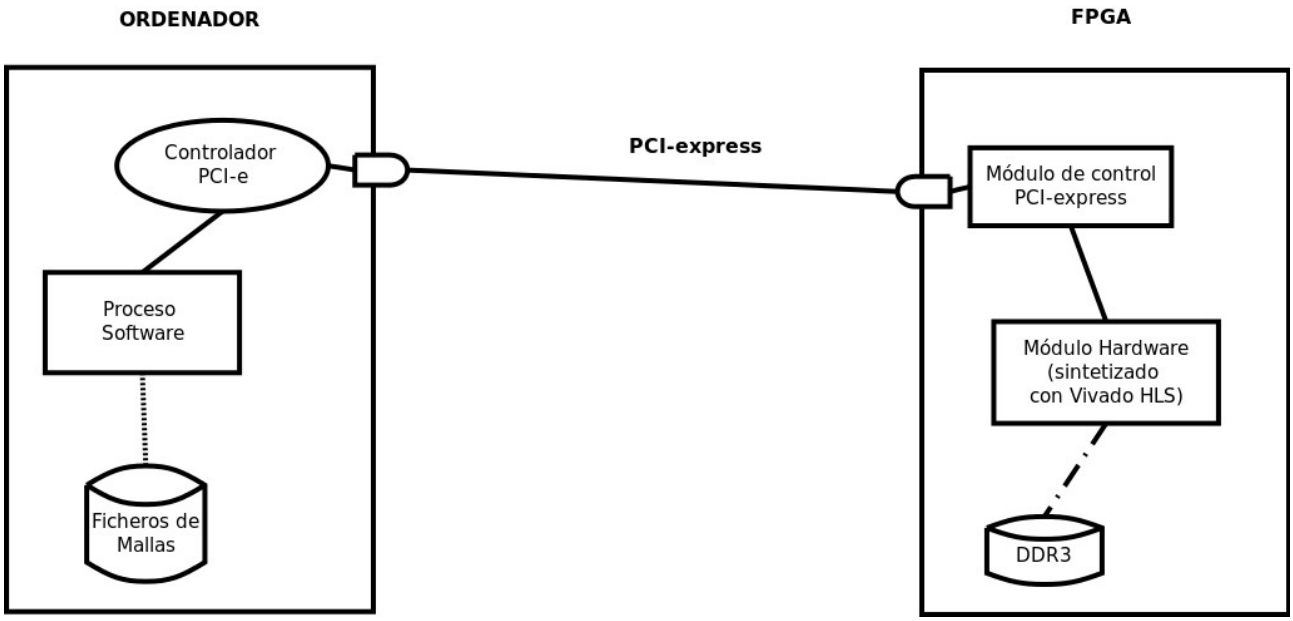

Figura 11- Diagrama de bloques del diseño general. En él se pueden ver representados los procesos en que se subdivide el solver, así como los principales elementos del diseño que serán necesarios a priori (un controlador para el PCI-e tanto del lado del ordenador como del de la FPGA, así como la memoria externa.

Antes de entrar con las secciones siguientes, en las que se explicará con detalle cómo se manejan e interconectan cada uno de estos elementos, vamos a hacer un breve adelanto de las herramientas que ocupan el lugar de los controladores, mencionados de forma abstracta en nuestro análisis previo.
Para gestionar el control del bus PCI-express utilizaremos un driver, desarrollado por Northwest, que proporciona tanto una API a nivel de procesador, como un core hardware a nivel de la FPGA, que permiten manejar las comunicaciones a través de dicho bus de una forma cómoda. El funcionamiento de este driver está descrito un poco más adelante, en la subsección dedicada a las comunicaciones a través del bus PCI-express.

En cuanto al manejo de la memoria DDR3, requerirá del uso de un controlador especial, llamado MIG, cuyo uso describiremos también en las secciones sucesivas.
Una vez detallado todo esto, pasamos a describir los elementos que componen el diseño.

6.2.1 Interfaces entre los dispositivos

Un aspecto que es importante mencionar es el tipo de interfaz que se ha elegido en cada caso a la hora de conectar los tres elementos fundamentales que van dentro de la FPGA: el core de Northwest, la memoria DDR3, y el bloque de procesamiento.
Entre el core de Northwest y la memoria DDR3 la interfaz es de tipo AXI, lo mismo que sucede con la interfaz entre el bloque de usuario y la memoria DDR3. Como ya comentamos, este tipo de interfaz tiene la ventaja de que permite realizar lecturas y escrituras en ráfaga, lo que da la opción al usuario de acceder (o escribir) a una gran cantidad de información en poco tiempo, siempre que esta información esté escrita en posiciones contiguas de la memoria. Puede encontrarse una descripción detallada de la interfaz de tipo AXI en la sección 4.2.1.

En cuanto a la interfaz que conecta el core de Northwest con el diseño de usuario, esta es de tipo AXI4-Stream. Este tipo de interfaz obliga a realizar accesos en modo FIFO (esto es, la información llega al receptor en el mismo orden que se envió), lo cual es menos eficiente que la lectura/escritura en ráfaga. Sin embargo, el manejo de este tipo de interfaz es más sencillo, y teniendo en cuenta que la información que se pasan el core de Northwest y el bloque de usuario es bastante marginal (el número de puntos y aristas, y otros datos puntuales), no hay necesidad de utilizar una interfaz más pesada. Puede encontrarse información más detallada sobre las interfaces de tipo AXI4-Stream en la sección 4.2.2.

6.3 Comunicaciones a través del bus PCI-express

Ya se ha comentado en varias ocasiones que las comunicaciones entre la FPGA y procesador se realizan a través del bus PCI-express, cuyo control se realiza mediante un driver desarrollado por Northwest, tal y como se mencionó al comienzo de la sección anterior. Dicho driver proporciona una API para manejar el driver a nivel software, y un core IP para manejar el driver a nivel hardware. Ambos elementos, así como su manejo, se describen en las subsecciones siguientes.

6.3.1 API del driver DMA

En la sección anterior se mencionó este controlador, que proporciona una API que permite un fácil acceso al bus PCI-express. Dicha API dispone de varios métodos, cuyo uso nos ha permitido realizar lecturas y escrituras en diferentes modos, atendiendo a las distintas necesidades en cada momento. Algunos de los métodos proporcionados por la API son los siguientes:

DoMem: Este método permite acceder de forma directa a una posición de memoria, tanto en lectura como en escritura. Es menos eficiente que los métodos de envío y recepción de paquetes, por escribir bit a bit, pero será necesario tanto para gestionar los puertos de control del bloque hardware como para escribir directamente en la memoria DDR.

SetupPacketMode: Este método nos permite establecer el tipo de paquetes que van a enviarse a través del DMA. En particular, nos interesa el modo FIFO, que nos permitirá enviar los streams de datos a través del bus PCI-express.

PacketSendEx: Este método envía los paquetes de datos a través del driver. El método esperará a que la FPGA lea los paquetes para retornar el control al programa, que permanecerá parado mientras tanto.

PacketReceiveEx: Este método nos permite leer los paquetes que llegan a través del driver. Cuando se invoca, toma el control del proceso, quedando en espera hasta que llega un paquete. El método nos dirá el tamaño del paquete recibido.

PacketReturnReceive: Este método libera la memoria del pool de memoria reservado para los paquetes. Dicha memoria quedaría de otro modo bloqueada, provocando que a la larga la API sea incapaz de continuar.

Las funciones descritas en la sección anterior pueden ser utilizadas directamente dentro del código C, enlazando adecuadamente las librerías necesarias. Dentro del código implementado, se ha creado un bloque aparte, que contiene todos los aspectos relativos al manejo de estas funciones, de tal modo que este quede aislado del resto del código. En particular, la llamada a todas estas funciones se realiza en el nivel más bajo dentro del bloque software.
A la hora de transmitir la información entre el bloque software (en el PC) y el bloque hardware (dentro de la FPGA), los métodos utilizados han sido el PacketSendEx y el PacketReceiveEx, métodos que permiten enviar el stream de datos en forma de paquetes. Además, para conocer el estado del procesamiento dentro de la FPGA, así como para indicarle al nivel hardware el momento de empezar a realizar el procesamiento, se han utilizado tres señales (start, idle, y done), que se escriben y leen del BAR0, mediante el método DoMem.
La escritura directa en la memoria DDR3 no se describe en esta sección, pero se basa en un procedimiento similar.

6.3.2 Core de Northwest

El core de Northwest es un módulo prediseñado que se encarga de manejar la conexión de la FPGA con el bus PCI-express. Dicho core recibe toda la información que llega a través del bus, y se encarga también de enviar la información a través del mismo. Para que sea posible manejar dicha información de forma cómoda dentro de la FPGA, este core ofrece varias interfaces, entre las cuales hay dos interfaces AXI4-Stream (una denominada card to system, que maneja la información que viaja desde la FPGA hacia el exterior; y otra denominada system to card, que maneja el flujo de información en el sentido contrario), y una interfaz AXI.

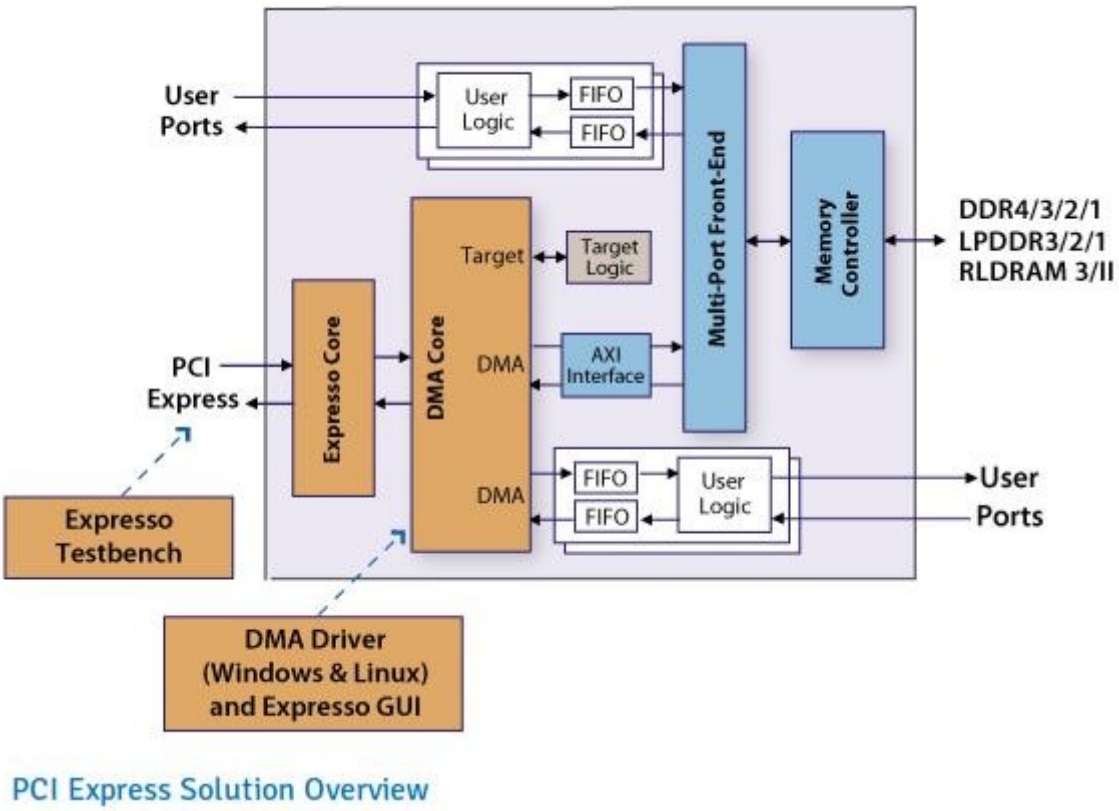

Figura 12 - Detalle del core de northwest a nivel de la FPGA. Imagen tomada de la página web de Northwest Logic, nwlogic.com/products/pci-express-solution/

El core de Northwest suministra también un reloj, que funciona a 250 MHz, con el que clockear el resto del diseño. Dicho reloj puede ser utilizado para alimentar otros módulos del diseño, o para generar otro reloj que funcione a una frecuencia más apropiada. El manejo del mismo se describe en la sección siguiente.

6.4 Comunicaciones dentro de la FPGA

Dentro de la FPGA, el manejo del bus PCIe se realiza mediante el uso de un core específico, desarrollado por Northwest para este propósito. Dicho core proporciona una interfaz AXI4-Stream, que utilizaremos para comunicarnos con el diseño Verilog generado por Vivado HLS a partir de nuestro código fuente (esto es, con el que en secciones previas se denominó bloque de procesamiento). Se puede encontrar una descripción del protocolo AXI4-Stream en la sección 4.2.2.

6.4.1 Comunicación entre el core de Northwest y el bloque de procesamiento. Clocking

A continuación, explicamos cómo se conecta el core de Northwest con el bloque de procesamiento que hemos sintetizado mediante la herramienta Vivado HLS. En un principio, podría parecer que basta con conectar directamente las interfaces AXI4-Stream de ambos bloques. Sin embargo, durante el desarrollo del mismo se vio que la conexión directa era insuficiente, ya que el diseño, una vez conectado, no funcionaba. Esto es

debido a que el core de Northwest funciona a una frecuencia de 250 MHz, mientras que el bloque de procesamiento generado por Vivado HLS funcionaba únicamente a 100 MHz. En primera instancia, se intentó conseguir que el diseño HLS funcionase a 250 MHz, modificando las opciones de síntesis dentro de la herramienta, pero esta era incapaz de generar un diseño que funcionase a dicha frecuencia, por lo que hubo que optar por otra solución, que fue la siguiente: se introduce un generador de reloj dentro de la FPGA, que genere un reloj de 100 MHz a partir del reloj de 250 MHz que proporciona el core de Northwest, y se introducen dos FIFOs (una para la interfaz s2c y otra para la interfaz c2s) entre nuestro bloque de procesamiento y el core de Northwest, de modo que cada bloque procese a la frecuencia que le interesa. El detalle de este diseño puede encontrarse en la figura 13.

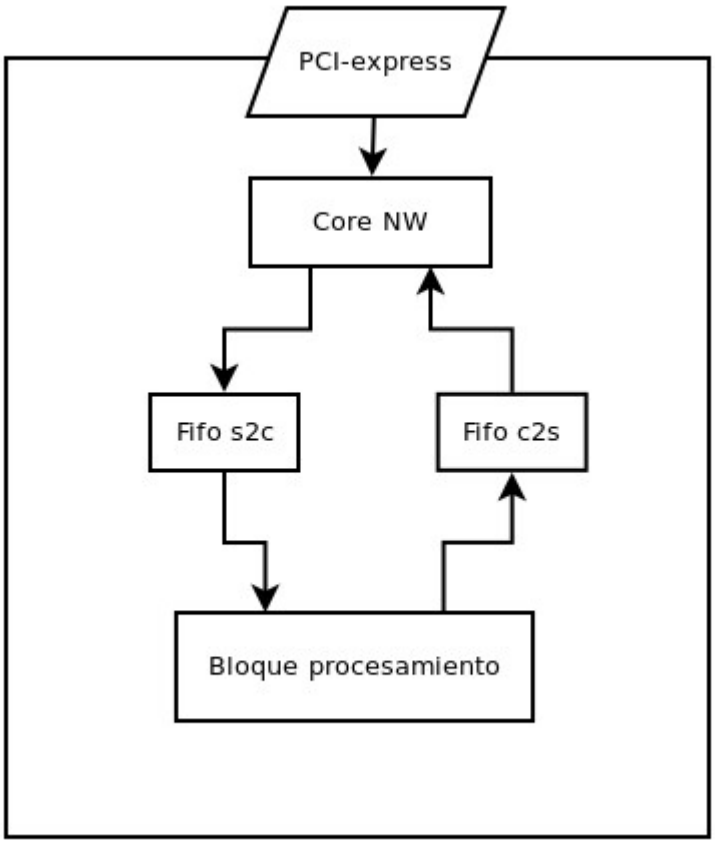

Figura 13 - Detalle de la conexión entre el bloque de procesamiento y el core de Northwest (dentro de la FPGA)

6.4.2 Manejo de la memoria. El módulo MIG

El manejo de la memoria DDR3 se realiza a través de un core prediseñado, denominado MIG. Este core nos permite seleccionar algunas opciones de configuración para la memoria, además de darnos distintas opciones relativas a su interfaz. Explicamos detalladamente algunas de estas opciones.

Tipo de memoria (y de componentes de la memoria): El módulo MIG permite controlar

varios tipos diferentes de memorias, no sólo la DDR3, por lo que debemos indicarle el tipo de memoria y el tipo de módulos que la componen (SODIMM).

Frecuencia de funcionamiento de la memoria: El MIG nos permite escoger la frecuencia a la que funcionará la memoria, en un rango limitado. Nosotros hemos escogido hacer funcionar la memoria a 800 MHz.

Reloj de entrada al MIG: El mig debe ser alimentado con dos señales de reloj, una principal y otra de referencia, que deben estar a la misma frecuencia. En este caso, hemos indicado al MIG que los relojes de entrada funcionarán a 400 MHz (la mitad de la frecuencia de la memoria).

Reloj UI: El MIG nos proporciona una señal de reloj, que funcionará a una frecuencia proporcional a la frecuencia interna de la memoria. En este caso, le hemos indicado que la proporción sea de 4:1, por lo que nos dará un reloj de 200 MHz para alimentar nuestro diseño.

Interfaz AXI: Al configurar el MIG, tenemos la opción de pedirle al diseño que nos proporcione una interfaz de tipo AXI, opción que hemos seleccionado, ya que nos facilitará el manejo del dispositivo desde nuestro diseño.

Pinout: Aunque la DDR está integrada en la placa, no forma parte en sí misma de la FPGA, por lo que es necesario indicarle al MIG los pines que debe utilizar para comunicarse con la misma. El pinout nos lo ha proporcionado un diseño de ejemplo que hay en la página de Xilinx (www.xilinx.com).

Una vez seleccionadas todas estas opciones, el core generator nos dará un diseño que podremos utilizar para manejar la memoria DDR3. La interfaz proporcionada puede verse en la figura 14. Toda la documentación relativa a este dispositivo puede encontrarse en [8].

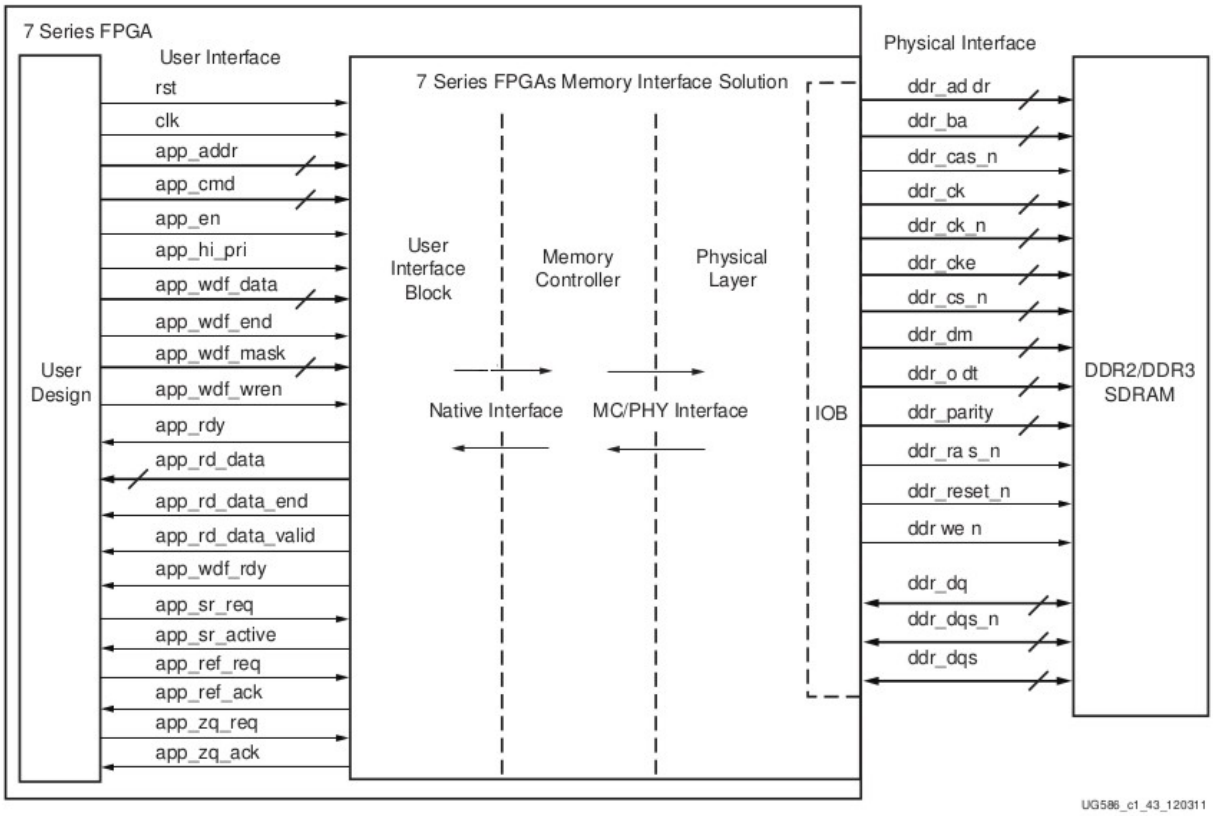

Figura 14 - Arquitectura del MIG. Figura extraída de [8]

6.4.3 Comunicación entre el bloque de procesamiento y el MIG

La comunicación entre el bloque de procesamiento y la memoria se lleva a cabo a través de un canal AXI, mediante el cual pueden realizarse lecturas y escrituras en ráfaga. El extremo maestro está del lado del bloque de procesamiento, de tal modo que es este el que dirige las comunicaciones, leyendo de la memoria o escribiendo en ella cuando lo necesite.

Dado que el MIG funciona a 200 MHz y el bloque de procesamiento lo hace a 100 MHz, ha sido necesaria la inclusión de varios dispostivos entre ambos, entre ellos dos FIFOs y un AXI clock converter. La arquitectura de esta conexión puede verse en detalle en la figura 15.

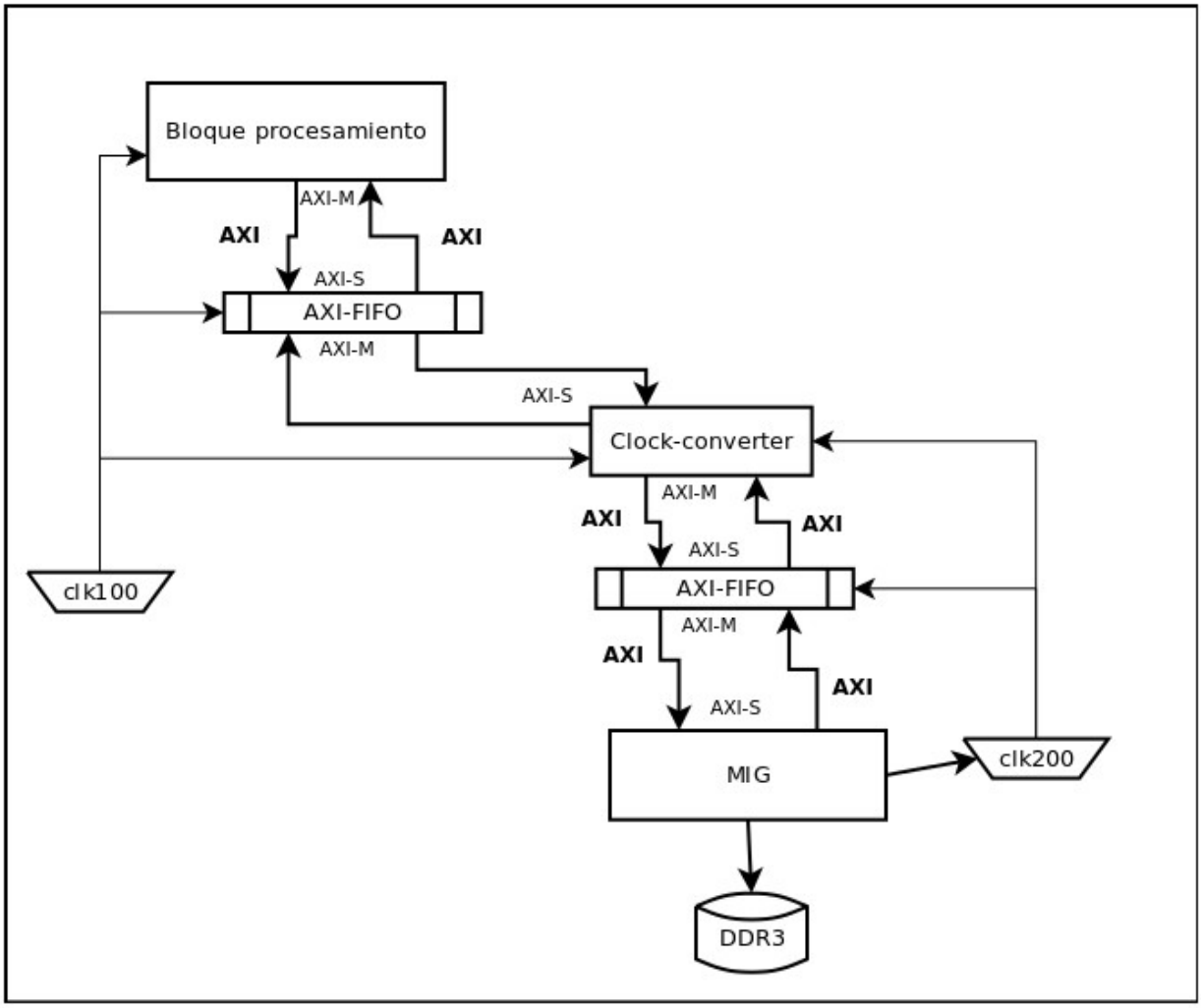

Figura 15 - Interconexión entre el bloque de procesamiento y el MIG.

6.4.4 Comunicación entre el core de Northwest el MIG

Para que sea posible escribir la malla que se va a procesar en la memoria DDR3 sin que esta pase por el bloque de procesamiento, es necesario que el core de Northwest tenga acceso directo al MIG. Esto es posible gracias a que el core de Northwest dispone de una interfaz AXI maestra, la cual está sin utilizar en nuestro diseño previo.
En este punto puede detectarse una dificultad, y es que la interfaz AXI del MIG ya está siendo utilizada por el bloque de procesamiento. Esto va a obligar a ambos cores a compartir dicha interfaz, tal y como se describe en la sección siguiente.

6.4.5 Compartición del acceso al MIG

En las secciones anteriores se especificó que el MIG comunica tanto con el bloque hardware como con el core de Northwest vía su interfaz AXI. Dado que el MIG dispone únicamente de una interfaz de este tipo, será necesario insertar un nuevo elemento,

también previamente diseñado, que permita compartir el acceso a este dispositivo. Dicho elemento es un AXI Interconnect, un core IP proporcionado por la herramienta de síntesis (Vivado), que permite que varias interfaces maestras estén conectadas simultáneamente a una esclava, arbitrando el acceso a la segunda.
El AXI interconnect insertado en el diseño dispone, por sí mismo, de dos interfaces esclavas (conectadas a cada una de las interfaces maestras del bloque de procesamiento y del core de Northwest respectivamente) y una interfaz maestra (conectada al MIG). El detalle de esta parte del diseño puede encontrarse en la figura 16.

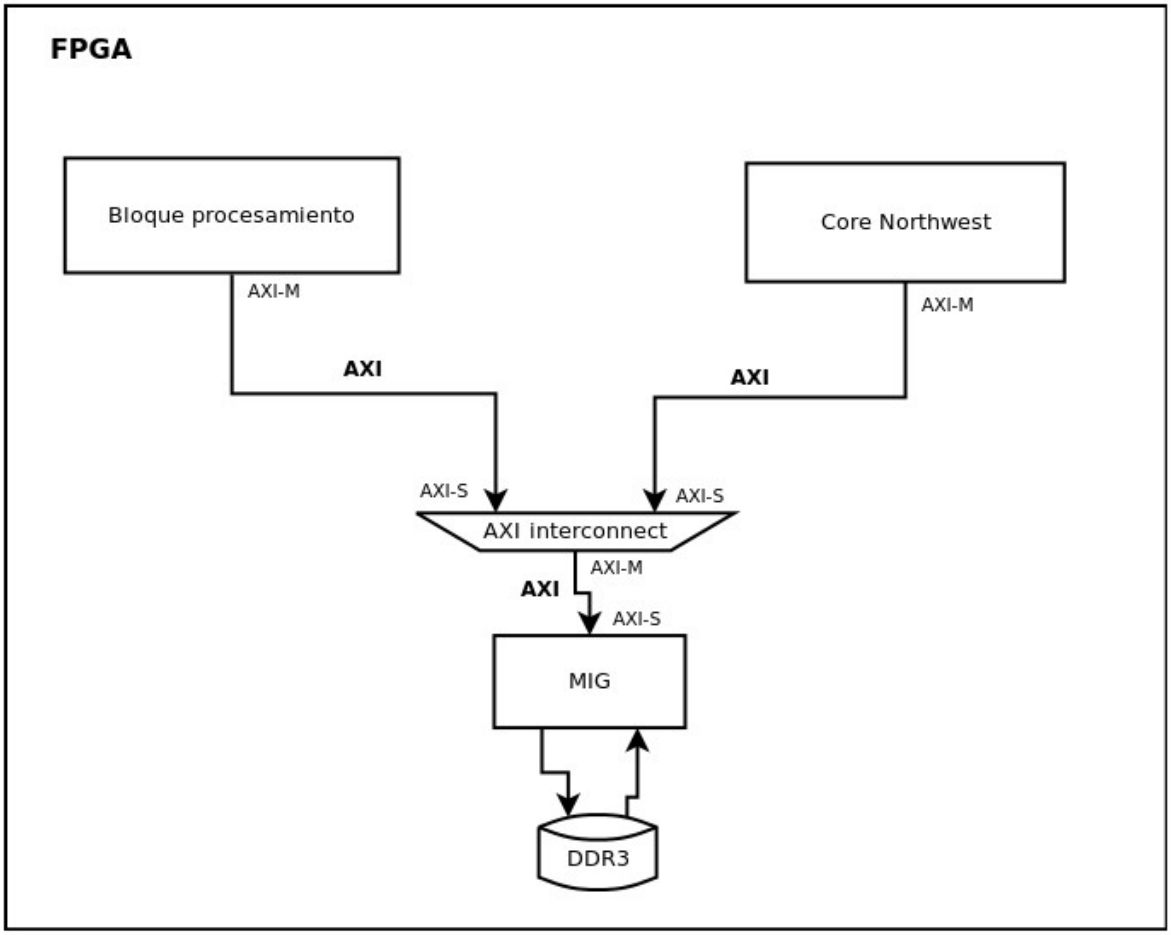

Figura 16 - Detalle de la compartición del puerto AXI del MIG entre el bloque de procesamiento y el core de Northwest (no se muestran las FIFOs entre el bloque de procesaiento y el MIG)

7. Validación y resultados

Una vez completada la implementación del solver heterogéneo, llega la hora de validar su funcionamiento y obtener los resultados de área y rendimiento.
La validación consiste, fundamentalmente, en garantizar que el solver heterogéneo que hemos construido obtiene los mismos resultados que el solver CFD de partida. Obviamente, si queremos acelerar un solver, en el proceso su funcionamiento no debería verse alterado, y eso es lo que vamos a comprobar.
En cuanto a los resultados obtenidos, la finalidad es comprobar qué grado de aceleración hemos conseguido (si es que hemos conseguido acelerar algo). No hay que olvidar que el objetivo ulterior del trabajo era acelerar el solver CFD que teníamos originalmente, por lo que, si no se ha conseguido tal cosa, el resultado será negativo.
En las subsecciones siguientes se presentan los métodos de validación y resultados obtenidos en mayor detalle.

7.1 Métodos de validación

A la hora de validar si el solver implementado funciona correctamente, se han utilizado una serie de mallas de distinto tamaño, con formato legible por el TAU, que es, en la actualidad, el sistema de referencia dentro del mundo de los CFDs.
Dichas mallas se han probado previamente en el solver utilizado como punto de partida (descrito en la sección 3.1), guardando los resultados. Una vez realizada esta prueba, se realiza la prueba equivalente en el solver heterogéneo implementado, verificando en todo momento que los resultados obtenidos en nuestro solver y los obtenidos con el solver original son exactamente los mismos.
Las pruebas realizadas abarcan tanto mallas con estructura multidominio como mallas con un solo dominio, ya que el solver debe estar preparado para responder tanto al multidominio como a mallas simples.

A continuación se presentan las mallas utilizadas como elemento de prueba, junto con sus características principales.

Nombre: param_md_coarse_0
Número de puntos: 2609
Número de caras: *7755*
Dominios: 3

Nombre: param_sd_coarse_0
Número de puntos: 2609
Número de caras: *7755*
Dominios: 1

7.2 Pruebas realizadas

En primer lugar, una vez el sistema heterogéneo se ha implementado y funciona adecuadamente, se anotan los valores arrojados por las herramientas de síntesis, en lo relativo a área utilizada, así como a cumplimiento de restricciones temporales del sistema.

En segundo lugar, se han realizado una serie de ejecuciones con las mallas proporcionadas, calculando el tiempo de respuesta del solver software de partida para utilizarlo como referencia. A continuación, se programa la FPGA con el solver implementado, y se repiten las mismas pruebas para el solver heterogéneo implementado, anotando los tiempos de respuesta obtenidos en este segundo caso. El objetivo, en última instancia, es obtener una relación del rendimiento obtenido por nuestra implementación con respecto a la implementación del solver de partida.

7.3 Resultados obtenidos

En esta subsección se presentan los resultados obtenidos durante las diferentes pruebas, y también los resultados del proceso de síntesis, tanto en la síntesis de alto nivel (Vivado HLS) como en la de bajo nivel (Vivado).

7.3.1 Optimización del rendimiento

La herramienta de síntesis de alto nivel (Vivado HLS) nos permite aplicar una serie de directivas para optimizar el rendimiento del sistema. Además, nos permite estimar el número de ciclos que tardará el sistema, si le indicamos el tamaño de los bucles que no dependan de una constante.
En esta sección presentamos los resultados estimados para una malla de 5000 puntos, 3000 caras, y 3 dominios.
En primer lugar, se presentan los resultados sin aplicar ninguna directiva de optimización. Estos pueden verse en la figura 17:

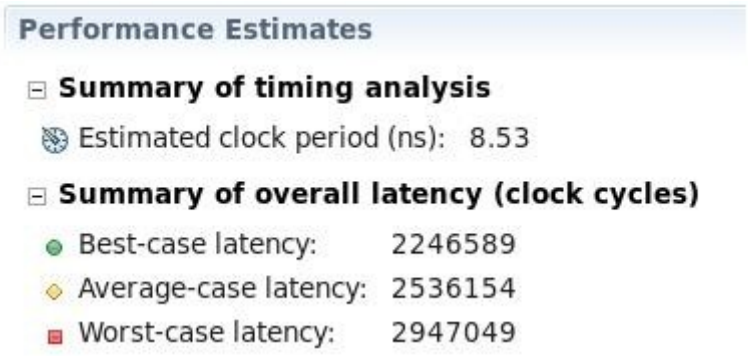

Figura 17 - Valores de latencia y periodo estimado de reloj para el diseño sin optimizar

En la figura aparecen cuatro valores, organizados en dos grupos. El primer valor, *Estimated clock period*, representa el periodo de reloj estimado del diseño. Esto nos dice a qué frecuencia trabajará la FPGA (en este caso, 8.53 ns, en torno a 100 MHz, que era el objetivo marcado). Los otros tres valores nos dan el caso mejor, peor, e intermedio, del número de ciclos de reloj totales que tardará el diseño en ejecutarse (en este caso, entre 2,2 y 2,9 millones de ciclos).
Realizando un cálculo sencillo, podemos obtener una estimación sencilla del tiempo de ejecución de nuestro diseño, sabiendo que cada ciclo tarda unos 10 ns, y que hay 2,5

millones de ciclos:

10^{-8} s/ciclo x 2.5 * 10^{6} ciclos = 2.5 * 10^{-2} seg

En la práctica, veremos que el programa tarda un poco más.

A continuación, presentamos la estimación de tiempos arrojada por la herramienta una vez insertadas las directivas de optimización. Esta puede verse en la figura 18.
En este caso, el tiempo de respuesta medio estimado es el siguiente:

10^{-8} s/ciclo x 8.2 * 10^{5} ciclos = 8.2 * 10^{-3} seg

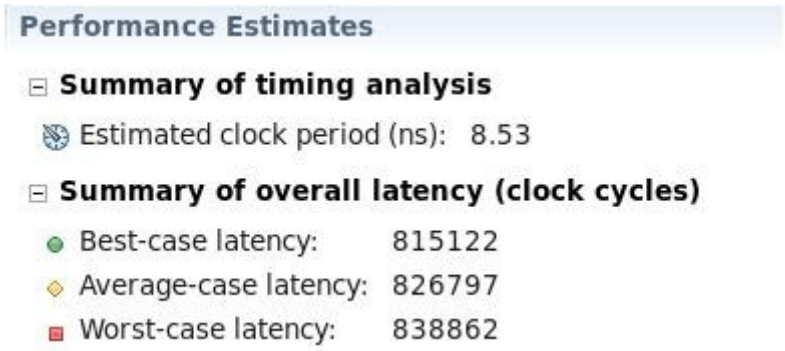

Figura 18 - Valores de latencia y periodo estimado de reloj para el diseño optimizado

Como podemos ver, el diseño optimizado reduce el tiempo de ejecución aproximadamente a un tercio del original. Esto en la práctica no será así, ya que el coste de la lectura y escritura de las mallas no está siendo contemplado.

7.3.2 Tiempos obtenidos

En esta sección se presentan los tiempos obtenidos tras la ejecución del algoritmo con las mallas mencionadas en la sección 5.1, tanto para el solver de partida como para el solver hardware. Estos tiempos nos permiten tener una idea más clara del grado de aceleración obtenido. Cada ejecución se realiza 100 veces, mostrándose también la varianza obtenida en los resultados.

Solver	**Malla**	**Tiempo medio de ejecución (s)**	**Varianza**
Solver software	param_sd	0.35	2.8 * 10^{-5}
Solver software	param_md	0.42	2.2 * 10^{-5}
Solver heterogéneo	param_sd	0.28	3.3 * 10^{-4}
Solver heterogéneo	param_md	0.25	9.8 * 10^{-5}

Como puede verse en la tabla, el solver implementado consigue reducir aproximadamente a la mitad el tiempo de ejecución del algoritmo. Este resultado no es muy exhaustivo, ya

que las pruebas se han realizado con pocas mallas, pero en principio cabe esperar que se mantenga con otro tipo de mallas, dado el determinismo del proceso.

7.3.3 Resultados de área

La síntesis del bloque de procesamiento dentro de la herramienta Vivado HLS nos arroja los siguientes resultados de área:

Area Estimates

Summary

	BRAM_18K	DSP48E	FF	LUT	SLICE
Component	30	487	82963	128609	-
Expression	-	-	0	926	-
FIFO	-	-	-	-	-
Memory	1547	-	0	0	-
Multiplexer	-	-	-	1465	-
Register	-	-	1923	-	-
ShiftMemory	-	-	-	-	-
Total	1577	487	84886	131000	0
Available	2060	2800	607200	303600	75900
Utilization (%)	76	17	13	43	0

Tabla 1 – Estimación de área arrojada por la herramienta Vivado HLS

Por otra parte, tras la síntesis con la herramienta Vivado (síntesis de bajo nivel), los resultados de área vienen dados por la siguiente tabla:

Slice Logic

Site Type	Used	Loced	Available	Util%
Slice LUTs*	174023	0	303600	57.31
LUT as Logic	158681	0	303600	52.26
LUT as Memory	15342	0	130800	11.72
LUT as Distributed RAM	2894	0		
LUT as Shift Register	12448	0		
Slice Registers	150404	2	607200	24.77
Register as Flip Flop	150404	2	607200	24.77
Register as Latch	0	0	607200	0.00
F7 Muxes	1353	0	151800	0.89
F8 Muxes	493	0	75900	0.64

Tabla 2 – Cálculo del área ocupada tras la síntesis en de bajo nivel (Vivado)

Estos resultados son bastante convenientes, ya que en nuestro código habíamos reservado espacio suficiente para procesar dominios de un tamaño considerable, y aún

con ello disponemos de suficiente área como para instalar el resto del diseño.
En este punto es importante tener en cuenta la necesidad de que sobre área, ya que de cara a trabajos futuros, nuestra intención sería introducir solvers más complejos, lo que requerirá de la utilización de mayor superficie dentro de la FPGA.

7.3.4 Cumplimiento de restricciones

Las restricciones de tiempo de un sistema son una serie de restricciones que se imponen durante la fase de diseño, que hacen referencia a los dominios de reloj que deben imperar en cada región del mismo. El cumplimiento de estas restricciones no siempre se produce a rajatabla, pudiendo sufrir el sistema algún retraso que merme su rendimiento. Si se produce una violación de las restricciones que alcance un grado suficientemente elevado, el sistema corre el riesgo de desincronizarse, dejando de funcionar por completo (y pudiendo llegar, incluso, a bloquear por completo la máquina cuando se invoca el solver).

En nuestro caso, las restricciones de tiempo no se han cumplido, pero el fallo entra dentro de los márgenes aceptables, siendo el peor SLACK de 2.25 ns.

8. Conclusiones

En esta sección se presentan las conclusiones tras el trabajo realizado, incluyendo un análisis de los resultados obtenidos, de la tecnología utilizada, y de las limitaciones de este tipo de sistemas.

8.1 Análisis de los resultados

Los resultados obtenidos son, en el mejor de los casos, mímimamente aceptables. El grado de aceleración obtenido (un 2x) es mínimo, quedando muy lejos de lo que sería un objetivo industrial. Hay que tener en cuenta que el solver que hemos acelerado no es un solver industrial, sino una versión más sencilla, susceptible, además, de ser optimizada mediante otro tipo de técnicas (como puede, por ejemplo, ser la paralelización mediante OpenMP), que fácilmente pueden alcanzar estos niveles de optimización. Considerando esta cuestión, y valorando el esfuerzo de desarrollo de este sistema en comparación con el esfuerzo que habrían implicado otro tipo de técnicas, es difícil obtener una conclusión positiva del resultado obtenido.
Pese a ello, los resultados no son del todo negativos. Si se hace un análisis del número de ciclos de reloj necesarios para la ejecución del solver, antes y después de aplicar las optimizaciones, podemos observar que la mejora que se consigue con la herramienta Vivado HLS es mínima, mostrándose una cierta mejora en el rendimiento, muy pequeña en comparación con lo que serían los objetivos industriales.

La conclusión que obtenemos de este análisis es que la propia herramienta, que nos debía proporcionar los métodos necesarios para paralelizar el código, no ha dado los resultados esperados. Debido al escaso control que nos proporciona la misma, resulta muy difícil mejorar este rendimiento. Este hecho choca con los resultados obtenidos en trabajos previos, en los que se utilizaban herramientas que, aunque resultan menos amigables, proporcionan un grado de control mucho mayor sobre el código, pudiendo alcanzar grados de optimización mucho mayores. Así pues, si bien es cierto que el resultado no es bueno, esto puede achacársele, en gran medida, a la herramienta, y por tanto cabe esperar que de cara a un futuro este tipo de técnicas acaben siendo una forma válida de acelerar algoritmos CFD.

8.2 Análisis del trabajo realizado

Del trabajo realizado, la parte que mayor carga se ha llevado ha sido la integración de todo el diseño a nivel hardware (el diseño generado con Vivado HLS, los cores IP utilizados, etcétera). Durante la realización de estas tareas, que en apariencia no deberían haber consumido tanto tiempo (ya que la tecnología utilizada está en general muy probada, salvo Vivado HLS), ha sido el cuello de botella del proyecto en todo momento, resultando muy costosa, con una curva de aprendizaje muy alta debido a la escasa documentación y a la falta de experiencia en el manejo de algunos de estos dispositivos.
En cambio, el aspecto que a priori debería haber sido principal en el proceso, que era la optimización del código con Vivado HLS, ha resultado, finalmente una de las tareas que menos tiempo han llevado, aunque sin duda aún había cierto margen de mejora en este aspecto.
En general, ha resultado muy difícil alcanzar el grado de profundidad necesario en la tarea

de optimización, lo que probablemente ha motivado el que los resultados no hayan resultado todo lo buenos que habría sido deseable.

8.3 Limitaciones de la tecnología

Tras el trabajo realizado, observamos algunas limitaciones importantes en la tecnología utilizada, que hacen que el futuro de este tipo de herramientas como metodología de aceleración sea, cuando menos, cuestionable.

La primera (y más importante) limitación reside en el tamaño de la FPGA. Si bien el método que hemos optimizado es explícito, y resulta posible dividir la malla en trozos para su procesamiento. Sin embargo, muchos de los métodos de estado del arte dentro del mundo de los CFDs son implícitos. Estos métodos requieren de la inversión de una matriz compuesta por los puntos de la malla, y su aceleración requeriría que la malla entera estuviese cargada en la FPGA, lo que es imposible.
Existen, en la actualidad, algunos métodos semiimplicitos que permiten acercar las aproximaciones implícita y explícita, perdiendo orden de convergencia a cambio de ganar velocidad de ejecución y reducir los requisitos de memoria. Estos métodos podrían ejecutarse en la FPGA, permitiendo así continuar con el trabajo.

La segunda limitación de este tipo de tecnologías reside en la posibilidad de aplicar limitadores de segundo orden. Este tipo de limitadores requieren del cálculo de los gradientes de un punto, lo que implica el acceso a vecinos de segundo orden (esto es, los vecinos de los vecinos) de los puntos. Esto complica las dependencias de datos, que hacen muy difícil la paralelización. Aunque esta dificultad no parece insalvable, lo cierto es que durante el desarrollo del trabajo no se ha visto una solución clara para la misma.

8.4 Limitaciones de las herramientas

Las limitaciones a la hora de construir este tipo de soluciones no residen únicamente en la tecnología, sino también en las herramientas. En particular, la herramienta Vivado HLS ha resultado ser bastante poco flexible, sobre todo a la hora de aplicar las optimizaciones en el código. Si bien es cierto que el que las optimizaciones se apliquen con solo insertar una directiva, esto no es bastante para lograr los objetivos que serían requisito para que el diseño pase el corte industrial. A menudo, cuando nos encontrábamos trabajando con la herramienta, observábamos que los intervalos de inicio de los bucles que habíamos paralelizado estaban muy lejos del objetivo pretendido, incluso si no existían dependencias entre los datos. Estos problemas, que aparentemente deberían haberse resuelto mediante la inserción de otras directivas (como las directivas de dependencia o de inline) a menudo persistían, impidiéndonos alcanzar el rendimiento que habría sido deseable. La imposibilidad de entrar a trabajar con mayor nivel de detalle nos obligaba, en estos casos, a conformarnos con el rendimiento obtenido, o como mucho, a intentar jugar con los parámetros de las directivas en busca de una mejora en el rendimiento. Por si fuese poco, a menudo la introducción de directivas causaba fallos en el funcionamiento del sistema, o pérdidas considerables en el número de ciclos obtenidos, que no parecían responder a ninguna causa razonable, sino más bien a un fallo de la propia herramienta. Todas estas cuestiones, en suma, hacen que nuestra confianza en la herramienta haya mermado considerablemente, haciendo que, de cara a trabajos futuros, nos planteemos el

uso de otras herramientas, menos amigables pero que ofrecen un grado de control mucho mayor.

9. Trabajo futuro

De cara al futuro, hay una amplia gama de opciones abiertas. Considerando que el solver implementado está construido en base a una versión relativamente sencilla de los solvers ya existentes en el mundo CFD, un posible camino a explorar sería la implementación de un nuevo solver, basado ahora en versiones algo más complejas (que incluyan, por ejemplo, flujos viscosos; o que resuelvan las ecuaciones de Navier-Stokes).
Uno de los caminos más interesantes a investigar sería la implementación de un solver heterogéneo del tipo del presentado en este documento, que tenga como punto de partida un solver semiimplícito como los mencionados en la sección 2.1.1. Este tipo de solver resultaría más fácil de implementar que uno implícito (que plantea problemas técnicos bastante importantes, como se describió en la sección 3), y estaría mucho más cerca del estado del arte en el mundo CFD.

Además de todas estas opciones, sin duda interesantes, uno de los trabajos cuya realización resultaría imperativa de cara a que este tipo de tecnología pueda llegar a ser competitiva en el mundo de los CFDs sería la sustitución de la herramienta de síntesis de alto nivel (Vivado HLS) por alguna otra herramienta, como Impulse C, que aporte el grado de control necesario para poder aprovechar al máximo el potencial de paralelización que tienen este tipo de algoritmos (que es, por otro lado, muy grande).

Referencias

[1] – Sanchez-Roman D, Sutter G, Lopez-Buedo S, Gonzlez I, Gomez-Arribas F, Aracil J. ”An Euler solver accelerator in FPGA for computational fluid dynamics applications”,VII Southern Programmable Logic Conference, SPL2011 2011.

[2] – L. Xiao, X. Zhang, Z. Kuang, B. Feng, and J. Kang, “Auto- CFD: efficiently parallelizing CFD appli ications on clusters”, in Proc. IEEE Int Cluster Computing Co 2003, pp. 4653.

[3] – T. Brandvik and G. Pullan “Acceleration of a 3D Euler solver using commodity graphics hardware”, in 46th AIAA Aerospace Sciences Meeting and Exhibit, 2008.

[4] – AMBA AXI and ACE Protocol Specification. www.arm.com

[5] – AMBA 4 AXI4-Stream Protocol ® . Version: 1.0 Specification . www.arm.com

[6] – J. Blazek, “Computational fluid dynamics: Principles and applications”; Elsevier, 2001.

[7] – Ch. Hirsch, “Numerical computation of internal and external flows”; Butterworth-Heinemann , 2007.

[8] – “7 Series FPGAs Memory Interface Solutions v1.8 . User Guide”, www.xinilnx.com, ref ug586.

[9] – “VC707 Evaluation Board for the Virtex-7 FPGA User Guide ”, www.xinilnx.com, ref ug885.

[10] – “Vivado HLS User Guide (Xilinx)”, www.xilinx.com

[11] – “DMA Driver User Guide 2.26”, Northwest Logic.

[12] – Northwest Core Documentation, Northwest Logic.

[13] – “Heterogeneous parallel systems for accelerating simulations based on discrete grid numerical methods”, http://worldwide.espacenet.com/

[14] – A. Quarteroni, R. Sacco, F. Saleri; “Numerical Mathmatics”; Springer, 2000

[15] - V.G. Asouti et al., "Unsteady CFD Computations Using Vertex-Centered Finite Volumes for Unstructured Grids on Graphics Processing Units," Int'l J. Numerical Methods in Fluids,19 May 2010, doi:10.1002/fld.2352.